

Package: vellum (via r-universe)

October 10, 2026

Title Low-Level Graphics with Device-Independent Layout and Queryable Scenes

Version 0.7.0.9000

Description A grid-like low-level graphics system for R. Text metrics and layout are resolved without a graphics device, so one solved scene renders to PNG, SVG, or PDF with the same geometry, and the finished scene can report that geometry back element by element (a data key and a resolved device-pixel box) or be queried by point. The scene graph, unit/layout engine, and rendering run in a Rust backend.

License MIT + file LICENSE

URL <https://github.com/r-vellum/vellum>,
<https://r-vellum.github.io/vellum/>,
<https://schochastics.r-universe.dev/vellum>

BugReports <https://github.com/r-vellum/vellum/issues>

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 8.0.0.9000

Config/rextendr/version 0.5.0

SystemRequirements Cargo (Rust's package manager), rustc >= 1.92.0, xz

Depends R (>= 4.2)

Suggests jsonlite, testthat (>= 3.0.0), withr, ggplot2, knitr, png, base64enc, rmarkdown, xml2, magick

VignetteBuilder knitr

Config/testthat/edition 3

Imports cli, grDevices, grid, S7, systemfonts, textshaping, vctrs

Collate 'gpar.R' 'viewport.R' 'grob.R' 'api.R' 'analysis.R'
'animate.R' 'datashade.R' 'debug.R' 'device.R'
'extendr-wrappers.R' 'fonts.R' 'geometry2.R' 'lint.R'

'output.R' 'paint.R' 'pick.R' 'place.R' 'scene-model.R'
 'serialize.R' 'spread.R' 'style.R' 'text.R' 'unit.R'
 'vellum-package.R' 'zzz.R'

Config/pak/sysreqs libfontconfig1-dev libfreetype6-dev libfribidi-dev libharfbuzz-
 dev libicu-dev xz-utils libclang-dev

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

Date/Publication 2026-09-03 17:09:11 UTC

RemoteUrl <https://github.com/r-vellum/vellum>

RemoteRef HEAD

RemoteSha 51f18676989ec8cdb4c4a401a900106b247a0adb

Contents

as_mask	3
as_scene_spec	4
as_vellum	5
as_vellum_scene	6
datashade	7
datashade_lines	9
describe	12
display	12
dynspread	13
element_geometry	14
font_pin	16
gradients	17
grob	19
grobwidth	27
hit_test	28
md	29
node_names	30
pdf_pages	30
profile_render	31
render_all	32
scene_diff	33
scene_fonts	34
scene_hash	35
scene_inset	35
scene_model	37
scene_png	38
scene_raster	39
scene_stats	40
scene_svg	41
scene_write	41
sketch	42
spread	43
stroke_to_path	44

style	46
text_path_grob	49
vl_arrow	50
vl_buffer	51
vl_clear_render_cache	52
vl_contour	52
vl_convert	54
vl_empty_region	55
vl_gpar	56
vl_hatch	58
vl_hull	59
vl_lint	60
vl_lint_assert	62
vl_lint_finding	63
vl_lint_overlay	64
vl_lint_rule	65
vl_nearest	67
vl_path_op	68
vl_pattern	69
vl_place	70
vl_render_animation	72
vl_scene	75
vl_shadow	77
vl_strwidth	78
vl_svg_path	79
vl_unit	81
vl_viewport	83
why_size	85

as_mask	<i>Masks</i>
---------	--------------

Description

Wrap a grob (or list of grobs) as a mask for `vl_viewport(mask = ...)`. The mask content is rendered to an isolated layer; its coverage then modulates the visibility of the viewport's contents.

Usage

```
as_mask(grob, type = c("alpha", "luminance"))
```

Arguments

grob	A grob, or a list of grobs, drawn in the masked viewport's coordinate system.
type	"alpha" (default) uses the mask's opacity as coverage; "luminance" uses its brightness (white shows, black hides).

Value

A vellum_mask object.

Examples

```
as_mask(circle_grob(r = 0.4, gp = vl_gpar(fill = "white", col = NA)))
```

as_scene_spec	<i>Convert a scene to and from a plain list</i>
----------------------	---

Description

`as_scene_spec()` turns a rendered-ready scene into a plain nested list of R vectors, and `from_scene_spec()` rebuilds the scene from it. Together they make a compiled scene a *value you can store or send*, rather than something that only exists inside the session that built it.

Usage

```
as_scene_spec(scene)

from_scene_spec(spec)
```

Arguments

scene	A <code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method.
spec	A list from <code>as_scene_spec()</code> .

Details

This is one level below `vellumplot`'s plot spec. A plot spec is portable and re-renderable at any size; a scene spec is the *resolved* artifact — the grobs, units and viewports as they will actually be drawn. They compose: keep the plot spec to re-render, keep the scene spec to reproduce exactly this scene.

The conversion is generic over the S7 property model rather than written per grob type, so a grob added to vellum later serializes with no change here.

Value

`as_scene_spec()`: a nested list, with a `version` element. `from_scene_spec()`: a `vellum_scene`.

See Also

[scene_write\(\)](#), [scene_diff\(\)](#), [scene_hash\(\)](#)

Examples

```
s <- vl_scene(3, 2) |>
  draw(circle_grob(r = 0.3, gp = vl_gpar(fill = "tomato", col = NA)))
spec <- as_scene_spec(s)
spec$version
identical(as_scene_spec(from_scene_spec(spec)), spec)
```

as_vellum

Render grid graphics (ggplot2 / lattice / grid) through vellum

Description

`as_vellum()` converts a grid grob tree — or a ggplot2 plot — into a `vl_scene()` by letting an offscreen grid device resolve all coordinates to absolute units, then emitting vellum grobs. `render_grid()` does that and writes the result. This is the interop path (grid-based graphics rendered by vellum's deterministic backend); a native graphics device is future work.

Usage

```
as_vellum(x, width = 7, height = 7, dpi = 96, bg = "white")

render_grid(
  x,
  path,
  width = 7,
  height = 7,
  dpi = 96,
  bg = "white",
  text = c("native", "outline")
)
```

Arguments

<code>x</code>	A grid grob/gTree/gtable, or a ggplot object.
<code>width, height</code>	Page size in inches.
<code>dpi, bg</code>	As in <code>vl_scene()</code> .
<code>path</code>	Output file (.png/.svg/.pdf).
<code>text</code>	Passed to <code>render()</code> (SVG text mode).

Value

`as_vellum()`: a `vellum_scene`. `render_grid()`: `path`, invisibly.

Examples

```
## Not run:
library(ggplot2)
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
render_grid(p, "plot.png", width = 6, height = 4)

## End(Not run)
```

as_vellum_scene	<i>Coerce an object to a vellum scene</i>
-----------------	---

Description

The extensible seam a higher-level package (e.g. a grammar layer) implements to compile its own plot object into a `vl_scene()`. `render()` coerces its input through this generic, so `render(x, path)` works for any `x` that has an `as_vellum_scene()` method. An identity method for `vellum_scene` is provided.

Usage

```
as_vellum_scene(x, ...)
```

Arguments

<code>x</code>	An object to coerce: a <code>vellum_scene</code> , or a type a downstream package has taught to compile by defining an <code>as_vellum_scene()</code> method.
<code>...</code>	Passed on to methods.

Details

This is the stable *compiler-backend* entry point: downstream packages should target `as_vellum_scene()` (and the exported `grob/viewport/unit` constructors) rather than vellum's internal `compile()` / `.scene_to_backend()` helpers.

Value

A `vellum_scene`.

Examples

```
sc <- vl_scene()
identical(as_vellum_scene(sc), sc) # the identity method returns its input
```

datashade

Aggregate-then-shade a large point cloud (datashader-style)

Description

For data beyond the point where drawing one marker each is practical (overplotted, millions of points), the fast and *overplotting-honest* approach is not to draw markers faster but to **not draw markers at all**: bin the points into a canvas-sized grid in one pass, then colour each cell by its density. This is what makes **datashader** fast — aggregation decouples cost from both point count and overplotting. `datashade()` returns a single `raster_grob()` you draw like any other grob.

Usage

```
datashade(
  x,
  y,
  weight = NULL,
  width = 600L,
  height = 400L,
  xlim = NULL,
  ylim = NULL,
  category = NULL,
  colors = c("#deebf7", "#08306b"),
  how = c("eq_hist", "log", "cbrt", "linear"),
  span = NULL,
  clip = NULL,
  spread = NULL,
  interpolate = FALSE,
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)
```

Arguments

<code>x, y</code>	Point coordinates (plain numerics, in data space).
<code>weight</code>	Optional per-point weight; cells accumulate the summed weight instead of a plain count. <code>NULL</code> counts. A scalar is recycled to every point; otherwise <code>weight</code> must be the same length as <code>x</code> .
<code>width, height</code>	Aggregation grid size in cells (= output raster pixels).
<code>xlim, ylim</code>	Data range to bin over; default the finite range of <code>x/y</code> .
<code>category</code>	Optional per-point category (factor or vector) selecting categorical (<code>count_cat</code>) shading; see Details. <code>NULL</code> (default) shades by plain density.

<code>colors</code>	For density shading, two or more colours forming the low-to-high ramp. For categorical shading (<code>category</code> set), a per-category hue vector — named by category level, or one colour per level in level order.
<code>how</code>	Density-to-colour mapping: <code>"eq_hist"</code> (histogram equalisation — datashader's default, reveals structure across orders of magnitude), <code>"log"</code> , <code>"cbrt"</code> (cube root), or <code>"linear"</code> . Also drives the per-cell opacity under categorical shading.
<code>span</code>	Optional <code>c(lo, hi)</code> density values mapped to the ends of the colour ramp / opacity range; densities outside are clamped. <code>NULL</code> (default) uses the full observed range.
<code>clip</code>	Optional percentile pair in <code>[0, 1]</code> (e.g. <code>c(0.01, 0.99)</code>) deriving <code>span</code> from the quantiles of the non-empty cell densities — a robust way to keep a few extreme cells from flattening the rest. Overrides <code>span</code> .
<code>spread</code>	Optional post-aggregation spreading, applied to the shaded raster to keep sparse output visible (see <code>spread()</code> / <code>dynspread()</code>): <code>NULL</code> (default) none; a positive integer applies <code>spread()</code> with that pixel radius; <code>"auto"</code> applies <code>dynspread()</code> (radius chosen from the image density).
<code>interpolate</code>	Passed to <code>raster_grob()</code> ; <code>FALSE</code> keeps hard bin edges.
<code>name, vp, id, role</code>	Passed to <code>raster_grob()</code> (see <code>grob</code>).

Details

The points are binned over `xlim` x `ylim` into a `width` x `height` grid, so to line the image up with data axes draw it inside a `vl_viewport()` whose `xscale` / `yscale` match `xlim` / `ylim` (it fills the viewport, `npc 0..1`). For crisp bins make `width/height` match the viewport's pixel size and keep `interpolate = FALSE`.

Value

A `grob` (a raster), drawable with `draw()`.

Categorical shading (`count_cat`)

Pass `category` (a factor or vector, one value per point) to shade by *category* rather than plain density: each category is aggregated into its own count grid in the same single pass, and each cell is coloured by the **count-weighted average** of the category hues it contains, with opacity driven by the cell's total density (via `how`). This reveals which category dominates where *and* where categories mix, without overplotting bias. When `category` is set, `colors` is a per-category hue vector (named by level, or one colour per level in level order) instead of a low-to-high ramp.

See Also

`datashade_lines()` and `datashade_segments()` for the line/segment (dense-timeseries, network-edge) counterparts, and `dynspread()` / `spread()`.

Examples

```
set.seed(1)
n <- 1e6
x <- rnorm(n); y <- x * 0.5 + rnorm(n)
g <- datashade(x, y, width = 400, height = 300)
s <- vl_scene(6, 4.5) |>
  push(vl_viewport(xscale = range(x), yscale = range(y))) |>
  draw(g)

# Categorical: colour each cell by which group dominates it
grp <- sample(c("a", "b"), n, replace = TRUE)
gc <- datashade(x, y, category = grp, colors = c(a = "#e41a1c", b = "#377eb8"))
```

datashade_lines	<i>Aggregate-then-shade dense lines and segments (datashader-style)</i>
-----------------	---

Description

The line/segment analogue of [datashade\(\)](#). Past a few hundred thousand line vertices — a dense stack of timeseries, or the edges of a large graph — drawing each line as a vector primitive overplots into a solid mass and balloons the output. [datashade_lines\(\)](#) and [datashade_segments\(\)](#) instead **rasterise** the lines into a canvas-sized grid in one pass: each cell accumulates the (anti-aliased) coverage of the lines crossing it, so overlapping lines *add* and the grid records true line density. The grid is shaded exactly like [datashade\(\)](#) (colors/how/span/clip) and returned as a single [raster_grob\(\)](#).

Usage

```
datashade_lines(
  x,
  y,
  group = NULL,
  weight = NULL,
  width = 600L,
  height = 400L,
  xlim = NULL,
  ylim = NULL,
  colors = c("#deebf7", "#08306b"),
  how = c("eq_hist", "log", "cbrt", "linear"),
  span = NULL,
  clip = NULL,
  spread = NULL,
  interpolate = FALSE,
  name = NULL,
  vp = NULL,
  id = NULL,
```

```

    role = NULL
)

datashade_segments(
  x0,
  y0,
  x1,
  y1,
  weight = NULL,
  width = 600L,
  height = 400L,
  xlim = NULL,
  ylim = NULL,
  colors = c("#deebf7", "#08306b"),
  how = c("eq_hist", "log", "cbrt", "linear"),
  span = NULL,
  clip = NULL,
  spread = NULL,
  interpolate = FALSE,
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)

```

Arguments

x, y	For <code>datashade_lines()</code> , the polyline vertices (data space); a segment joins each consecutive pair.
group	For <code>datashade_lines()</code> , an optional per-vertex series id (factor or vector) the same length as <code>x</code> . The line breaks between vertices whose group differs, so multiple series pack into one call. <code>NULL</code> treats all vertices as one series (still broken by NA coordinates).
weight	Optional per-line weight (per start-vertex for <code>datashade_lines()</code> , per segment for <code>datashade_segments()</code>): cells accumulate summed weight instead of plain coverage. <code>NULL</code> weighs each line 1; a scalar is recycled; otherwise it must match the line/segment count.
width, height	Aggregation grid size in cells (= output raster pixels).
xlim, ylim	Data range to bin over; default the finite range of <code>x/y</code> .
colors	For density shading, two or more colours forming the low-to-high ramp. For categorical shading (<code>category</code> set), a per-category hue vector — named by category level, or one colour per level in level order.
how	Density-to-colour mapping: <code>"eq_hist"</code> (histogram equalisation — datashader's default, reveals structure across orders of magnitude), <code>"log"</code> , <code>"cbrt"</code> (cube root), or <code>"linear"</code> . Also drives the per-cell opacity under categorical shading.

<code>span</code>	Optional <code>c(lo, hi)</code> density values mapped to the ends of the colour ramp / opacity range; densities outside are clamped. <code>NULL</code> (default) uses the full observed range.
<code>clip</code>	Optional percentile pair in <code>[0, 1]</code> (e.g. <code>c(0.01, 0.99)</code>) deriving <code>span</code> from the quantiles of the non-empty cell densities — a robust way to keep a few extreme cells from flattening the rest. Overrides <code>span</code> .
<code>spread</code>	Optional post-aggregation spreading, applied to the shaded raster to keep sparse output visible (see <code>spread()</code> / <code>dynspread()</code>): <code>NULL</code> (default) none; a positive integer applies <code>spread()</code> with that pixel radius; <code>"auto"</code> applies <code>dynspread()</code> (radius chosen from the image density).
<code>interpolate</code>	Passed to <code>raster_grob()</code> ; <code>FALSE</code> keeps hard bin edges.
<code>name, vp, id, role</code>	Passed to <code>raster_grob()</code> (see <code>grob</code>).
<code>x0, y0, x1, y1</code>	For <code>datashade_segments()</code> , the segment endpoints (data space), one per segment; all four the same length.

Details

- `datashade_lines()` takes a **connected polyline**: a segment is drawn between each consecutive `(x, y)`. Pass `group` to pack several series into one call — the line breaks wherever the group changes; an `NA` in `x/y` also breaks it. This is the dense-timeseries path.
- `datashade_segments()` takes **independent segments** `(x0, y0) -> (x1, y1)`, one per element. This is the network-edge / `mark_segment` path.

Line coverage is anti-aliased (a Wu accumulator) and summed, so a line deposits roughly **weight** per cell it spans and dense bundles brighten honestly rather than saturating. As with `datashade()`, align the raster to data axes by drawing it in a `vl_viewport()` whose `xscale/yscale` match `xlim/ylim`.

Value

A `grob` (a raster), drawable with `draw()`.

See Also

`datashade()` for points; `dynspread()`/`spread()` for keeping thin lines visible.

Examples

```
set.seed(1)
# Dense timeseries: 400 random walks of 500 steps, packed into one raster.
k <- 400; m <- 500
walks <- apply(matrix(rnorm(k * m), m, k), 2, cumsum)
t <- rep(seq_len(m), k)
g <- datashade_lines(t, as.vector(walks), group = rep(seq_len(k), each = m),
                    width = 400, height = 300)

# Network edges: random segments shaded by edge density.
```

```
n <- 5000
e <- datashade_segments(rnorm(n), rnorm(n), rnorm(n), rnorm(n))
```

describe	<i>Set a scene's accessibility name and description</i>
----------	---

Description

Attach (or replace) an accessible **name** (**title**) and long **description** (**desc**, the alt text) on an existing scene. The SVG backend then emits `role="img" + <title>/<desc>`, and the PDF backend tags the page as a Figure with the description as Alt text — meeting WCAG 1.1.1 (text alternative). Equivalent to passing `title/``desc` to `vl_scene()`.

Usage

```
describe(scene, title = NULL, desc = NULL)
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> .
<code>title</code>	An accessible name (short), or <code>NULL</code> to leave unset.
<code>desc</code>	A long description / alt text, or <code>NULL</code> to leave unset.

Value

The scene, with the accessibility fields set (a new value).

Examples

```
vl_scene(2, 2) |>
  draw(points_grob(c(0.3, 0.7), 0.5, gp = vl_gpar(fill = "red"))) |>
  describe(title = "Two red dots", desc = "Two red points on a white field.")
```

display	<i>Display a scene in the active graphics device</i>
---------	--

Description

`display()` re-renders `scene` to fill the current graphics device and draws it there. Interactively this is the RStudio / Positron **Plots** pane; inside a knitr / Quarto chunk it becomes the chunk's figure (it draws to the chunk's device). This is the seam any package built on vellum can call to *show* output instead of writing a file: `scene` is coerced via `as_vellum_scene()`, so it also accepts e.g. a grammar's plot spec.

Usage

```
display(scene, ...)
```

Arguments

`scene` A `vl_scene()` or anything with an `as_vellum_scene()` method.
`...` Unused.

Details

To fill the window (no letterbox margins, like `ggplot2`) the scene is re-rendered at the device's size and pixel density, so its relative (`npc/native/layout`) content reflows to the window and absolute (`mm/in/pt`) content keeps its physical size. It draws through a grid grob that re-rasterizes on every draw, so **resizing the Plots pane re-renders the scene crisply** at the new size (round markers stay round) rather than stretching one bitmap. Use `render()` to write the scene at its *authored* width/height. Auto-printing a scene at the console (or calling `plot()` on it) displays it.

Inside a knitr / Quarto chunk the chunk's `dpi` option wins, so `knitr::opts_chunk$set(dpi = 200)` yields a genuine 200-dpi figure even on knitr's default `dev = "png"` device (which misreports its pixel density); outside knitting the scene's authored `vl_scene()` `dpi` is honored unless the live device reports a trustworthy higher density (e.g. a resized Plots pane).

Value

The (coerced) scene, invisibly.

Examples

```
## Not run:
vl_scene(4, 3) |>
  draw(circle_grob(r = 0.3, gp = vl_gpar(fill = "tomato", col = NA))) |>
  display()

## End(Not run)
```

`dynspread`

Dynamically spread a raster grob to a target density

Description

datashader's dynspread: pick the smallest spread radius (up to `max_px`) at which the shaded pixels become "connected enough" — the fraction of non-empty pixels with a non-empty neighbour reaches `threshold` — then `spread()` by it. Denser images spread less, sparse ones more, so a mix of dense and sparse regions all stay legible.

Usage

```
dynspread(
  grob,
  max_px = 3L,
  threshold = 0.5,
```

```

    shape = c("circle", "square"),
    how = c("over", "add")
  )

```

Arguments

grob	A raster grob (e.g. from datashade()). Other grob kinds are returned unchanged.
max_px	Largest spread radius to consider (non-negative integer).
threshold	Target fraction in (0, 1] of non-empty pixels that should have a non-empty neighbour; growth stops once it is reached.
shape	Neighbourhood shape: "circle" (default, Euclidean radius) or "square" (Chebyshev radius).
how	How overlapping spread pixels combine: "over" (default) keeps the colour of the most-opaque contributor; "add" accumulates opacity (clamped) so overlaps darken.

Value

A raster [grob](#) with the chosen spread applied.

See Also

[spread\(\)](#), [datashade\(\)](#), [datashade_lines\(\)](#).

Examples

```

set.seed(1)
g <- datashade_segments(rnorm(2000), rnorm(2000), rnorm(2000), rnorm(2000))
g2 <- dynspread(g)

```

element_geometry	<i>The true geometry of a scene's addressable elements</i>
------------------	--

Description

Returns the resolved device-pixel geometry of every **keyed** element: the two endpoints of a segment, the vertices of a line or polygon, the centre of a round mark, the opposite corners of a rect.

Usage

```
element_geometry(scene)
```

Arguments

scene A [vl_scene\(\)](#) (or anything with an [as_vellum_scene\(\)](#) method).

Details

This is what a host needs to hit-test *exactly* without asking the engine per event. `scene_model()` gives bounding boxes, which are right for a brush and misleading for diagonal or thin marks; this gives the shape the box was standing in for, once, so a browser can compute point-to-geometry distances locally at whatever rate it likes.

Value

A data frame of **key**, **kind**, **vertex** (1-based within the element), **ring** (1-based within the element), **x** and **y**. Zero rows if the scene has no keyed elements.

Reading the result

One row per vertex, grouped by **key**. What the vertices mean depends on **kind**:

kind	rows	meaning
segment	2	the endpoints
line, polygon	<i>k</i>	the vertices, in order; polygon is closed
path	<i>k</i>	all rings' vertices concatenated; ring separates them
point	1	the centre
rect	2	opposite corners
text, roundrect	2	the bounding box's corners

Coordinates are device pixels with y growing **down**, matching `scene_model()`'s boxes and the rendered SVG's coordinate system.

ring groups the vertices of a multi-ring **path** – a polygon with a hole, or a multipart feature – so a host can hit-test it the way the engine does. Without it the concatenated vertices are lossy: treating them as one closed ring invents a phantom edge from each ring's last vertex to the next ring's first. Split on **ring** and test the rings separately. Every other kind is a single ring, so **ring** is 1 throughout.

See Also

`vl_nearest()`, `scene_model()`

Examples

```
s <- vl_scene(4, 3, dpi = 96, bg = "white") |>
  draw(segments_grob(0.1, 0.1, 0.9, 0.9, key = "diagonal"))
element_geometry(s)
```

font_pin	<i>Pin a scene's fonts, and check them later</i>
----------	--

Description

`font_pin()` records the fonts a scene resolved to. `font_check()` re-resolves the same scene and reports what changed — a different file, a missing file, or a family that now resolves somewhere else.

Usage

```
font_pin(scene)
```

```
font_check(scene, pin, on_mismatch = c("warn", "error", "ignore"))
```

Arguments

scene	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method).
pin	A manifest from <code>font_pin()</code> .
on_mismatch	"warn" (default), "error", or "ignore" — what <code>font_check()</code> should do when the fonts have moved.

Value

`font_pin()`: a `vellum_font_pin` object. `font_check()`: invisibly, a data frame of differences (zero rows when everything matches), with a `status` of "changed", "missing" or "new".

Why this exists

vellum's determinism claim is that the same scene renders to the same pixels on every OS and in CI. Layout, shaping and rasterisation deliver that. Font *resolution* does not, and cannot: "sans" is a different file on macOS, Linux and Windows, and even the same family name can resolve to a different version of the same font.

So the claim holds only if the fonts are the same, and until now nothing checked. A pin turns "identical pixels" from an assumption into an assertion: record the manifest next to a reference image, and `font_check()` will say whether a pixel difference is your change or the machine's font stack.

What this does not do

It does not *make* fonts reproducible — it cannot install a font, and vellum deliberately resolves fonts through `systemfonts` so that it agrees with the rest of the R graphics ecosystem. Bundling font files with a scene would break that agreement and raises licensing questions vellum should not answer for you. The honest tool is a check, not a substitute.

The reliable fix, when you need it, is to register the exact file you mean with `systemfonts::register_font()` and pin *that*.

Examples

```
s <- vl_scene(4, 2) |> draw(text_grob("hello"))
pin <- font_pin(s)
pin
# In a test, next to a reference image:
nrow(font_check(s, pin)) == 0
```

gradients

Gradient fills

Description

Create a linear or radial gradient to use as a fill in `vl_gpar()`. A gradient interpolates between colour *stops*. Its geometry (`x1/y1/...` or `cx/cy/r`) is given in the coordinate system named by `units` and is resolved against the viewport at draw time, so the gradient transforms with the grob just like its outline.

Usage

```
linear_gradient(
  colours,
  stops = NULL,
  x1 = 0,
  y1 = 0,
  x2 = 1,
  y2 = 0,
  units = "npc",
  extend = "pad",
  interpolation = "srgb"
)
```

```
radial_gradient(
  colours,
  stops = NULL,
  cx = 0.5,
  cy = 0.5,
  r = 0.5,
  fx = cx,
  fy = cy,
  fr = 0,
  units = "npc",
  extend = "pad",
  interpolation = "srgb"
)
```

Arguments

colours	A vector of two or more colours (any R colour spec). With stops = NULL they are spread evenly across [0, 1].
stops	Optional offsets in [0, 1], one per colour. Defaults to evenly spaced.
x1, y1, x2, y2	Start and end points of a linear gradient (default a left-to-right sweep in npc).
units	Coordinate system for the geometry: one of "npc", "native", "mm", "in", "pt".
extend	How the gradient behaves outside [0, 1]: "pad" (clamp to the end stops), "repeat", or "reflect".
interpolation	Colour space the stops are blended in: "srgb" (default), "oklab" (perceptually uniform), or "oklch" (perceptual, hue-preserving). See Details.
cx, cy, r	Centre and radius of a radial gradient's <i>outer</i> circle — the end of the ramp (stop offset 1). Default centred, radius 0.5 npc.
fx, fy, fr	Centre and radius of the <i>focal</i> (start) circle — the origin of the ramp (stop offset 0). Defaults (fx = cx , fy = cy , fr = 0) give the ordinary concentric gradient; move fx/fy to place the highlight off-centre, or raise fr for an annular ramp between two circles. Radii must be non-negative.

Details

A radial gradient runs between two circles: the *focal* (start) circle **fx/fy/fr** at stop offset 0 and the *outer* (end) circle **cx/cy/r** at offset 1. By default they are concentric (**fx** = **cx**, **fy** = **cy**, **fr** = 0) — the classic centred highlight. Offsetting **fx/fy** moves the highlight off-centre (as for a sphere lit from one side); a non-zero **fr** gives an annular ramp between the two circles.

By default stops are blended in sRGB (each backend's native behaviour). Set **interpolation** = "oklab" to blend in the perceptually-uniform Oklab space instead, which removes the muddy, over-dark midtones and hue drift of sRGB blending — the ramp stays even and vivid. **interpolation** = "oklch" blends in the polar form of the same space (lightness, chroma, hue): hue and chroma move independently, so a ramp between two saturated colours keeps its chroma through the middle instead of dipping toward grey the way a straight line in Oklab can — at the cost of sweeping through the intermediate hues along the shorter arc (e.g. blue→yellow passes through green). All modes work identically on the raster, SVG, and PDF backends.

Value

A `vellum_gradient` object, suitable for `vl_gpar(fill = ...)`.

Examples

```
linear_gradient(c("white", "navy"))
linear_gradient(c("blue", "yellow"), interpolation = "oklab")
linear_gradient(c("blue", "yellow"), interpolation = "oklch")
radial_gradient(c("yellow", "red"), cx = 0.5, cy = 0.5, r = 0.5)
# off-centre highlight (a lit sphere): focal point up and to the left
```

```
radial_gradient(c("white", "navy"), cx = 0.5, cy = 0.5, r = 0.6,
               fx = 0.35, fy = 0.65)
```

grob

Graphical objects (grobs)

Description

Grobs are immutable value objects describing something to draw. Build them with the constructors below, add them to a scene with `draw()`, and render with `render()`. Coordinates accept a `vl_unit()` vector or a bare numeric (interpreted in the `default_units`, usually "npc").

Usage

```
rect_grob(
  x = 0.5,
  y = 0.5,
  width = 1,
  height = 1,
  sketch = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL,
  key = NULL,
  meta = NULL
)
```

```
roundrect_grob(
  x = 0.5,
  y = 0.5,
  width = 1,
  height = 1,
  r = 0.1,
  sketch = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL,
  key = NULL,
  meta = NULL
)
```

```
lines_grob(
```

```
x,  
y,  
arrow = NULL,  
start_cap = NULL,  
end_cap = NULL,  
offset = NULL,  
sketch = NULL,  
lwd_profile = NULL,  
gp = vl_gpar(),  
name = NULL,  
vp = NULL,  
id = NULL,  
role = NULL,  
key = NULL,  
meta = NULL  
)
```

```
polygon_grob(  
  x,  
  y,  
  sketch = NULL,  
  gp = vl_gpar(),  
  name = NULL,  
  vp = NULL,  
  id = NULL,  
  role = NULL,  
  key = NULL,  
  meta = NULL  
)
```

```
bezier_grob(  
  x,  
  y,  
  n = 60,  
  gp = vl_gpar(),  
  name = NULL,  
  vp = NULL,  
  id = NULL,  
  role = NULL  
)
```

```
spline_grob(  
  x,  
  y,  
  shape = 1,  
  n = 20,  
  open = TRUE,  
  gp = vl_gpar(),
```

```
    name = NULL,
    vp = NULL,
    id = NULL,
    role = NULL
)

circle_grob(
  x = 0.5,
  y = 0.5,
  r = 0.25,
  sketch = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL,
  key = NULL,
  meta = NULL
)

points_grob(
  x,
  y,
  size = vl_unit(2, "mm"),
  shape = "circle",
  sketch = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL,
  key = NULL,
  meta = NULL
)

hexagon_grob(
  x = 0.5,
  y = 0.5,
  size = vl_unit(2, "mm"),
  width = NULL,
  height = NULL,
  fill = NULL,
  orientation = c("flat", "pointy"),
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL,
```

```
    key = NULL,  
    meta = NULL  
)  
  
sector_grob(  
  x = 0.5,  
  y = 0.5,  
  r0 = 0,  
  r1 = 0.5,  
  theta0 = 0,  
  theta1 = 2 * pi,  
  fill = NULL,  
  arrow = NULL,  
  sketch = NULL,  
  gp = vl_gpar(),  
  name = NULL,  
  vp = NULL,  
  id = NULL,  
  role = NULL,  
  key = NULL,  
  meta = NULL  
)  
  
loop_grob(  
  x = 0.5,  
  y = 0.5,  
  size = vl_unit(4, "mm"),  
  foot = vl_unit(0, "mm"),  
  angle = 0,  
  width = 1,  
  arrow = NULL,  
  gp = vl_gpar(),  
  name = NULL,  
  vp = NULL,  
  id = NULL,  
  role = NULL  
)  
  
segments_grob(  
  x0,  
  y0,  
  x1,  
  y1,  
  arrow = NULL,  
  start_cap = NULL,  
  end_cap = NULL,  
  offset = NULL,  
  sketch = NULL,
```

```
gp = vl_gpar(),
name = NULL,
vp = NULL,
id = NULL,
role = NULL,
key = NULL,
meta = NULL,
col = NULL,
lwd = NULL
)

path_grob(
  x,
  y,
  id = NULL,
  rule = c("winding", "evenodd"),
  sketch = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  role = NULL,
  key = NULL,
  meta = NULL
)

raster_grob(
  image,
  x = 0.5,
  y = 0.5,
  width = 1,
  height = 1,
  interpolate = TRUE,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)

text_grob(
  label,
  x = 0.5,
  y = 0.5,
  just = "centre",
  rot = 0,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
```

```

    id = NULL,
    role = NULL,
    width = NULL,
    height = NULL,
    align = "left",
    fit = FALSE,
    key = NULL,
    meta = NULL
)

```

Arguments

<code>x, y</code>	Coordinates (<code>vl_unit()</code> or numeric).
<code>width, height</code>	Grob size (<code>vl_unit()</code> or numeric), recycled like <code>x/y</code> . For most grobs the drawn rectangle size. For <code>hexagon_grob()</code> , the optional per-hexagon full corner-to-corner extent along <code>x/y</code> : when both are given they override size (resolved per-axis, so a hexagon can be <i>non-regular</i> and tile a non-square lattice — e.g. "native" units tile in data space regardless of device aspect); a <i>regular</i> flat hexagon is <code>height == width * sqrt(3) / 2</code> ; leave both <code>NULL</code> to use circumradius size ; must be given together. For <code>loop_grob()</code> , <code>width</code> is instead a dimensionless lateral petal scale in <code>(0, 1]</code> (recycled per loop): <code>1</code> (default) is the full teardrop, smaller narrows the petal's waist without shortening it (the igraph "narrowing" factor). For <code>text_grob()</code> these mean something different: <code>width</code> is an absolute wrapping measure (<code>mm/cm/in/pt</code>), and giving it breaks the label into lines that fit, making the drawn block a box of exactly that width — so just anchors the box rather than the longest line. <code>height</code> is used only by <code>fit</code> . Relative units are rejected there on purpose: wrapping happens when the grob is built, and a viewport has no size in <code>npc/native</code> until render time.
<code>sketch</code>	Optional <code>sketch()</code> spec for a hand-drawn look; <code>NULL</code> = crisp.
<code>gp</code>	Graphical parameters, from <code>vl_gpar()</code> .
<code>name</code>	Optional name (for <code>edit_node()</code>).
<code>vp</code>	Optional <code>vl_viewport()</code> to draw this grob inside.
<code>id</code>	For most grobs, an optional semantic identifier emitted by the SVG backend as <code>data-vellum-id</code> (for interactivity, accessibility, and testing; ignored by raster/PDF). For path_grob only , <code>id</code> instead groups points (one value per point) into closed sub-paths: all points sharing an <code>id</code> form one sub-path (so a hole is a separate <code>id</code>), in first-appearance order (à la grid); <code>NULL</code> makes a single sub-path.
<code>role</code>	Optional ARIA role, emitted by the SVG backend as <code>role=</code> for accessibility (ignored by the raster and PDF backends).
<code>key</code>	Optional per-element data key(s) for the batched marks (<code>points_grob</code> , <code>circle_grob</code> , <code>rect_grob</code> , <code>segments_grob</code> , <code>hexagon_grob</code> , <code>sector_grob</code>), recycled to the element count like <code>fill</code> . Emitted by the SVG backend as <code>data-key</code> on each element and surfaced by <code>scene_model()</code> — the join

	key a host uses to map an interaction back to a datum. NULL (default) emits nothing (a static render is unchanged). Ignored by raster/PDF.
<code>meta</code>	Optional free-form per-element metadata for the batched marks: a list with one entry (record) per element (recycled), e.g. tooltip text or field values. It never reaches the backend (nothing drawn changes); it rides on the scene and is returned by <code>scene_model()</code> . NULL (default) = none.
<code>r</code>	Radius (<code>vl_unit()</code> or numeric).
<code>arrow</code>	An <code>vl_arrow()</code> spec to draw heads on the line/segment ends, or NULL for none.
<code>start_cap, end_cap</code>	Optional absolute-length <code>vl_unit()</code> s (mm/cm/ in/pt; a bare numeric is taken as mm) that shorten the drawn line inward from its start/end by that physical amount, resolved at render in device space — so the gap is exact at any size, dpi, and aspect ratio, with no reliance on the native scale. For <code>segments_grob()</code> the caps are per-element (scalar or length-n, recycled like the coordinates); for <code>lines_grob()</code> a single (scalar) cap trims each end of the whole polyline. NULL (default) leaves the endpoint untouched. When an <code>vl_arrow()</code> is also present its head is placed at the <i>capped</i> end, so the tip lands on the boundary (e.g. a node marker) rather than under it. This is what lets a directed edge stop at a node's radius. See the acceptance notes in the package for the degenerate cases (a cap $\geq$ the segment length draws nothing; a zero-length segment is skipped).
<code>offset</code>	Optional absolute-length <code>vl_unit()</code> (mm/cm/in/pt; a bare numeric is mm) that shifts the line perpendicular to its own direction by that physical amount, resolved at render in device space. The sign picks the side (+ left of the direction of travel, - right). For <code>segments_grob()</code> it is per-element (scalar or length-n) — passing a vector spreads parallel/reciprocal edges by a fixed physical spacing that tracks mm node sizes at any figure size; for <code>lines_grob()</code> a single (scalar) offset rigidly translates the whole polyline along the perpendicular of its overall direction. Applied before <code>start_cap/end_cap</code> and the arrowhead (offset, then cap, then head). NULL/0 (default) leaves the geometry untouched.
<code>lwd_profile</code>	For <code>lines_grob()</code> , an optional numeric vector of one width multiplier per vertex , turning the polyline into a variable-width stroke that tapers smoothly between vertices. Each value scales the resolved <code>lwd</code> , so <code>c(1, 2, 0)</code> runs from the nominal width, through double, to a point. Resolved at render , inside the grob's viewport, so the taper is correct at any figure size and moves with a panel that is laid out later — which is the difference from <code>stroke_to_path()</code> , whose outline is baked at a page size you supply. NULL (default) draws an ordinary uniform stroke, and is byte-for-byte the output you got before this argument existed. A varying-width stroke is generated as an outline and filled (no backend can stroke one path at several widths), so PNG, SVG and PDF all get the same geometry — it degrades nowhere. Two consequences: the stroke colour paints the ribbon while <code>lty</code> and dashing do not apply, and each ribbon is unique geometry, so it opts out of the batching fast paths — fine for the low element counts this is meant for, not for a million points.

	Cannot be combined with <code>sketch</code> . For a profile spread by arc length rather than per vertex, see <code>stroke_to_path()</code> 's <code>along</code> argument.
<code>n</code>	Number of points to sample the curve at (flattened to a polyline).
<code>shape</code>	Marker shape(s): "circle" (default), "square", "triangle", "diamond", "plus", "cross", "triangle_down", or "star", recycled per point. Filled shapes (all but "plus"/"cross") paint <code>gp\$fill</code> and outline with <code>gp\$col</code> , so an <i>open</i> marker is <code>fill = NA</code> with a <code>col</code> , a <i>filled</i> one sets <code>fill</code> ; "plus"/"cross" are stroke-only line glyphs.
<code>open</code>	If <code>FALSE</code> , the spline is closed (wraps end to start).
<code>size</code>	Loop extent: an absolute <code>vl_unit()</code> (mm/cm/in/pt; a bare numeric is mm), resolved to a device length at render so the loop tracks a node's mm size at any page size/dpi. Nested loops on one vertex pass growing <code>size</code> (same <code>x/y/angle</code>) for concentric teardrops. Recycled per loop.
<code>fill</code>	Per-element fill colour(s), recycled to the number of sectors. <code>NULL</code> falls back to <code>gp\$fill</code> .
<code>orientation</code>	Hexagon orientation: "flat" (default, flat top/bottom edge) or "pointy" (vertex at top). <code>size</code> is the circumradius (centre to vertex).
<code>r0, r1</code>	Inner and outer radius of each sector (<code>vl_unit()</code> or numeric; numeric is treated as "native"). <code>r0 = 0</code> gives a pie slice; <code>r0 == r1</code> gives an arc outline (stroke only, no fill).
<code>theta0, theta1</code>	Start and end angle of each sector, in radians , with 0 at 3 o'clock and increasing counter-clockwise.
<code>foot</code>	Node radius the loop's two feet attach at (an absolute <code>vl_unit()</code> ; 0 = both feet at the vertex, like <code>igraph</code>). A positive <code>foot</code> puts the feet on the node's boundary so the loop visibly leaves and re-enters the node edge, and a directed loop's head lands on the boundary rather than under the marker. Recycled per loop.
<code>angle</code>	Outward direction of the loop in radians (which way the teardrop bulges away from the vertex, e.g. away from the layout centroid).
<code>x0, y0, x1, y1</code>	Segment start/end coordinates (<code>vl_unit()</code> or numeric).
<code>col, lwd</code>	<code>segments_grob()</code> only — per-segment stroke colour and width, recycled to the segment count, mirroring the per-element <code>fill</code> that <code>hexagon_grob()</code> and <code>sector_grob()</code> carry. <code>NULL</code> (default) uses the shared <code>gp</code> , and the batch is drawn in one combined stroke exactly as before. Reach for these instead of building one grob per segment: varying width or colour that way costs an R-side grob each, which is the dominant expense on scenes made of many small marks. Note that no backend can stroke segments differently in a single call, so each segment is stroked on its own here — the saving is in R, not in the output size. These vary width <i>per element</i> , so the width steps between segments and the joins show. For width varying smoothly <i>along</i> one line, with no joins to give it away, see <code>lwd_profile</code> in <code>stroke_to_path()</code> .
<code>rule</code>	Fill rule: "winding" (non-zero, default) or "evenodd".

<code>image</code>	A raster image. Either a <code>grDevices::as.raster()</code> -compatible object — a matrix/array of colours or greyscale values, or a raster — or a path to a PNG file , decoded in the Rust backend so no R image package is needed. A numeric RGB/RGBA array in <code>[0, 1]</code> (what <code>png::readPNG()</code> returns) takes a fast path that skips the per-pixel colour-string round-trip <code>as.raster()</code> would otherwise do; the pixels are identical either way.
<code>interpolate</code>	Smoothly interpolate when scaling (default <code>TRUE</code>)? <code>FALSE</code> keeps hard pixel edges.
<code>label</code>	Character string(s) to draw.
<code>just</code>	Justification: <code>c(hjust, vjust)</code> as names (" <code>left</code> ", " <code>centre</code> ", " <code>right</code> ", " <code>bottom</code> ", " <code>top</code> ") or numbers in <code>[0, 1]</code> .
<code>rot</code>	Rotation in degrees, counter-clockwise.
<code>align</code>	Alignment of lines within the box: " <code>left</code> " (default), " <code>centre</code> "/" <code>center</code> ", " <code>right</code> ", or " <code>justify</code> " (flush both edges, last line of each paragraph left as-is).
<code>fit</code>	Auto-fit sizing. <code>FALSE</code> (default) keeps <code>gp\$fontsize</code> . <code>TRUE</code> shrinks the font — never grows it — until the wrapped block fits <code>width × height</code> , down to a floor of 4 pt; a number sets that floor instead. Requires <code>width</code> .

Details

`sector_grob()` draws a batch of annular sectors (pie / donut / rose wedges) in a single call. `gp$fill` recycles per sector; `gp$col/lwd` give a uniform stroke.

Passing `r0 == r1` gives an **open arc** (stroke only). Combined with an absolute (mm) radius at a "`native`" centre and an `vl_arrow()`, the radius is resolved to a device length at render (like a marker `size`), so the arc tracks an mm size at any page size or dpi; the arrowhead sits tangent to the outer arc's end. (For node-link **self-loops**, prefer `loop_grob()` — a teardrop, not a ring.)

`loop_grob()` draws **self-loops** for node-link diagrams as an igraph-style cubic **Bézier teardrop**: it leaves the vertex (`x, y`) (a "`native`" anchor), bulges out to `size` along `angle`, and returns, with an optional `vl_arrow()` head tangent to the curve at the returning foot. `size` and `foot` are absolute and resolved to device px **at render**, so the loop is a fixed physical size that scales with the mm node markers — no native-per-mm estimation, exact at any figure size/dpi.

Value

A grob object.

`grobwidth`

Size a unit by a grob's extent

Description

`grobwidth(grob)` and `grobheight(grob)` return a `vl_unit()` equal to the drawn width/height of `grob` — handy for sizing a `vl_viewport()` or `grid_layout()` track to its contents (e.g. a margin to an axis label). The extent is measured **eagerly** to absolute millimetres at construction, so it is exact for text and absolute-unit (`mm/in/pt`) grobs. A grob sized in `npc/native` has no viewport-independent extent and is measured against a fixed reference, so for those prefer `npc/native` directly.

Usage

```
grobwidth(grob, mult = 1)

grobheight(grob, mult = 1)
```

Arguments

<code>grob</code>	A grob (or composite subtree) to measure.
<code>mult</code>	A multiplier on the measured extent (default 1).

Value

A unit (in millimetres).

Examples

```
grobwidth(text_grob("A wide axis label", gp = vl_gpar(fontsize = 14)))
grobheight(rect_grob(height = vl_unit(8, "mm")))
```

hit_test

Hit-test a scene

Description

Find the topmost node drawn under a point. The scene is compiled into a colour pick-buffer (each grob drawn in a colour encoding its id, respecting clipping and paint order) using the same code path that renders it, so the result is geometry-, clip- and overlap-exact by construction. Markers and text are matched by their bounding box; lines and segments by a small pick band. grid ships no equivalent: `grid.locator()` waits for one interactive click and returns coordinates rather than a node.

Usage

```
hit_test(scene, x, y, units = c("npc", "px"))
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> .
<code>x, y</code>	Query point, in <code>units</code> : "npc" (default; the page, 0..1 with y up) or "px" (device pixels, top-left origin, y down).
<code>units</code>	"npc" or "px".

Value

The hit node's `name` (character); `NA_character_` if the topmost grob there is unnamed; or `NULL` if nothing is drawn at the point.

<code>md</code>	<i>Rich-text labels (markdown subset)</i>
-----------------	---

Description

`md()` builds a styled label from a small markdown/HTML-free subset, for use as the `label` of `text_grob()` (and anywhere a label is measured with `grobwidth()/grobheight()`). The base font/size/colour come from `gp`; markup spans override per run.

Usage

```
md(text)
```

Arguments

<code>text</code>	A markup string (or a character vector for per-element labels).
-------------------	---

Details

Supported markup:

- **`**bold**`**
- *`*italic*`* or `_italic_`
- ^{`^sup`} (superscript) and _{`~sub~`} (subscript)
- `[text]{#c00}` — a coloured span (any R colour: name or hex)

Spans nest (e.g. `**a2**`). `md()` with no markup is equivalent to the plain string. Embedded newlines (`\n`) start a new line (stacked baseline-to-baseline).

`md()` is vectorised: a length-1 input returns a single `vellum_md_label`; a longer vector returns a list of them (one per element), so a `vellumplot` mark can carry a per-datum rich label.

Value

A `vellum_md_label` (length-1 `text`) or a list of them (length > 1).

Examples

```
lab <- md("R^2 = **0.91**")
labs <- md(c("*a*", "**b**")) # a list of two labels
```

node_names	<i>Inspect and edit a scene by node name</i>
------------	--

Description

`node_names()` lists the names in a scene. `get_node()` returns the first node with a given name. `edit_node()` returns a new scene with that node's properties updated (copy-on-modify).

Usage

```
node_names(scene)

get_node(scene, name)

edit_node(scene, name, ...)
```

Arguments

scene	A <code>vl_scene()</code> .
name	A node name (set via the <code>name</code> argument of a grob/viewport).
...	Properties to set, e.g. <code>gp = vl_gpar(col = "red")</code> .

Value

`node_names()`: character. `get_node()`: a node. `edit_node()`: a `vellum_scene`.

pdf_pages	<i>Write several scenes as the pages of one PDF</i>
-----------	---

Description

`render()` writes one page. This writes a document: a report's worth of figures, one facet per page, an animation as a contact sheet.

Usage

```
pdf_pages(scenes, path = NULL)
```

Arguments

<code>scenes</code>	A list of scenes (each a <code>vl_scene()</code> or anything with an <code>as_vellum_scene()</code> method).
<code>path</code>	Output file. NULL returns the bytes instead.

Details

Pages may differ in size — each carries its own page box — so a landscape figure can sit between two portrait ones.

Tagging follows each page's own metadata, exactly as it does for a single-page render (see `vignette("accessible-output")`): the pages are drawn by the same code, so a document cannot drift from a single-page file.

Value

`path` invisibly, or a raw vector when `path` is NULL.

See Also

`render()` for a single page.

Examples

```
pages <- lapply(c("tomato", "steelblue", "seagreen"), function(col) {
  vl_scene(4, 3, dpi = 96, bg = "white") |>
    draw(circle_grob(r = 0.3, gp = vl_gpar(fill = col)))
})
f <- tempfile(fileext = ".pdf")
pdf_pages(pages, f)
file.size(f) > 0
```

`profile_render`

Where a scene spends its render time

Description

Attributes render cost to the individual marks that caused it, and splits the three phases a render goes through: **build** (constructing the R value), **compile** (the R→Rust replay, including text shaping), and **raster** (drawing). Turns "this plot is slow" into a specific answer.

Usage

```
profile_render(scene, reps = 3)
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method.
<code>reps</code>	Number of render repetitions to time; the median is reported.

Details

Read the phase split first. Compile time is R-side — S7 property validation, grob construction, unit records — and on scenes with many small grobs it routinely dominates, in which case no amount of backend tuning will help. The per-node table only accounts for raster time.

Timing is armed only for this call, so ordinary renders pay nothing for it.

Value

A data frame ordered by cost, with columns **kind**, **name**, **n** (elements), **seconds** and **pct** (share of raster time). Rows are aggregated per mark — a vectorised **text_grob()** of 200 labels compiles to 200 nodes but reports as one row — so the table reads as "which of my marks is expensive". The phase split is attached as the **"phases"** attribute and shown when printed.

See Also

[vl_lint\(\)](#), [scene_stats\(\)](#)

Examples

```
set.seed(1)
s <- vl_scene(4, 3) |>
  draw(points_grob(runif(2000), runif(2000))) |>
  draw(text_grob("title", y = 0.95, gp = vl_gpar(fontsize = 16)))
profile_render(s)
```

render_all

Render many scenes in parallel

Description

Renders a list of independent scenes across cores. This is embarrassingly parallel — one whole scene per worker, nothing shared — which is why it exists while *tiling a single raster* across threads does not: that needs synchronised access to one pixmap and is declined in **PERFORMANCE.md**.

Usage

```
render_all(scenes, paths, workers = NULL, ...)
```

Arguments

scenes	A named or unnamed list of scenes.
paths	Output paths, one per scene. The format of each comes from its extension, exactly as in render() . When scenes is named and paths is a single directory, files are named after the list.

workers	Number of parallel workers. Defaults to one per available core, capped at the number of scenes.
...	Passed to <code>render()</code> for every scene.

Details

The saving is real when the scenes are substantial and there are several of them. For a handful of small figures the process/thread overhead dominates and a plain `lapply(scenes, render)` is as fast; this reports enough to tell.

Value

The paths written, invisibly.

See Also

`render()`, `pdf_pages()`

Examples

```
scenes <- list(
  a = vl_scene(3, 2, dpi = 96) |> draw(circle_grob(gp = vl_gpar(fill = "tomato"))),
  b = vl_scene(3, 2, dpi = 96) |> draw(rect_grob(gp = vl_gpar(fill = "steelblue")))
)
render_all(scenes, file.path(tempdir(), c("a.png", "b.png")))
```

<code>scene_diff</code>	<i>What changed between two scenes</i>
-------------------------	--

Description

Compares two scenes structurally and reports the differences in terms of the scene — “the node `axis-x` moved”, “3 marks changed fill” — rather than as a pixel diff.

Usage

```
scene_diff(a, b, max_depth = 40L)
```

Arguments

a, b	Two scenes (<code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method).
max_depth	Stop descending after this many levels (guards a runaway report on deeply nested trees).

Details

This is a better basis for visual-regression testing than comparing rendered images, for one specific reason: an image diff is sensitive to the font stack, so the same code compared across two machines produces differences that swamp the real change. A structural diff compares what was *asked for*, and is unaffected.

Value

A data frame with columns `path` (where in the tree), `change` ("added", "removed", or "changed"), and `detail`. Empty when the two scenes are equivalent. Printing shows a readable summary.

See Also

`scene_hash()` for a yes/no answer, `as_scene_spec()`.

Examples

```
a <- vl_scene(3, 2) |> draw(circle_grob(r = 0.3, gp = vl_gpar(fill = "red")))
b <- vl_scene(3, 2) |> draw(circle_grob(r = 0.4, gp = vl_gpar(fill = "blue")))
scene_diff(a, b)
```

<code>scene_fonts</code>	<i>Which fonts a scene actually used</i>
--------------------------	--

Description

Reports the font **files** a scene's text resolved to, read off the shaped glyphs rather than re-resolved from family names — so it says what was used, not what would be picked if asked again.

Usage

```
scene_fonts(scene)
```

Arguments

`scene` A `vl_scene()` (or anything with an `as_vellum_scene()` method).

Value

A data frame with `path`, `index` (the face within a font collection), `glyphs` (how many glyphs came from that face), `file` (the basename) and `exists`. Zero rows if the scene has no text.

See Also

`font_pin()` to record this and check it later.

Examples

```
s <- vl_scene(4, 2) |> draw(text_grob("hello", gp = vl_gpar(fontfamily = "serif")))
scene_fonts(s)
```

scene_hash
A content fingerprint for a scene

Description

A stable hash of everything that defines the scene — geometry, style, page size, background. Two scenes that would render identically hash identically; any change to what is drawn changes the hash.

Usage

```
scene_hash(scene)
```

Arguments

scene A `vl_scene()`, or anything with an `as_vellum_scene()` method.

Details

Useful for caching (has this scene changed since I last rendered it?) and for asserting in tests that a refactor did not alter the scene.

Value

A length-1 character hash.

See Also

`scene_diff()` to see *what* changed, `as_scene_spec()`.

Examples

```
a <- vl_scene(2, 2) |> draw(circle_grob(gp = vl_gpar(fill = "red")))
b <- vl_scene(2, 2) |> draw(circle_grob(gp = vl_gpar(fill = "blue")))
scene_hash(a) == scene_hash(a)
scene_hash(a) == scene_hash(b)
```

scene_inset
Inset one scene inside another

Description

Places a whole scene inside a region of another one. Because a scene is a retained tree with a known resolved size, this is a graft rather than a re-render: the guest's grobs are spliced under a viewport of the host, and the result is an ordinary scene you can keep building on, edit by name, or inset again.

Usage

```
scene_inset(
  host,
  guest,
  x = 0.5,
  y = 0.5,
  width = 0.3,
  height = 0.3,
  name = NULL,
  ...
)
```

Arguments

<code>host</code>	The scene to place into.
<code>guest</code>	The scene to place. Its page size, background and dpi are <i>not</i> carried over — it becomes a region of the host, which owns those.
<code>x, y, width, height</code>	The region, in the host's current coordinates.
<code>name</code>	Optional name for the region's viewport, so the inset can later be found with <code>node_names()</code> and edited with <code>edit_node()</code> .
<code>...</code>	Further arguments to <code>vl_viewport()</code> (<code>clip</code> , <code>angle</code> , <code>gp</code> , <code>mask</code> , <code>blur</code> , <code>shadow</code> , ...).

Details

This is the mechanism a composition layer needs — inset maps, small multiples assembled from independently-built plots, a legend built separately and dropped into a corner. The *policy* (should panel edges align across composed plots? should axes be shared?) belongs above vellum, in a grammar; what the engine provides is the ability to nest one scene in another at all.

The guest's `npc` coordinates become relative to the region it is placed in, so a guest built on a square page and inset into a wide region will stretch. Match the aspect ratio, or build the guest at the aspect you want.

Value

A `vellum_scene`.

See Also

`vl_viewport()`, `as_scene_spec()`

Examples

```
main <- vl_scene(4, 3) |>
  draw(rect_grob(gp = vl_gpar(fill = "#eef2f6", col = NA)))
mini <- vl_scene(1, 1) |>
```

```
draw(circle_grob(r = 0.4, gp = vl_gpar(fill = "tomato", col = NA)))
scene_inset(main, mini, x = 0.8, y = 0.75, width = 0.3, height = 0.35)
```

scene_model	<i>A serializable, per-element model of a scene</i>
-------------	---

Description

`scene_model()` walks a rendered scene and returns one row per drawn element of the *keyable* marks (points, circles, rects, hexagons, sectors, segments), pairing each element's grammar-supplied identity — its data **key**, free-form **meta**, grob **id/name**, and enclosing **panel** — with its resolved device-pixel bounding box. It is the host-agnostic bridge underlying interactivity: a host renders the SVG (each element tagged with **data-key**, see [scene_svg\(\)](#)) and uses this table to map an event back to the originating datum.

Usage

```
scene_model(scene)
```

Arguments

scene A [vl_scene\(\)](#) (or anything coercible via [as_vellum_scene\(\)](#)).

Details

Elements are returned in paint order. **key/meta** are **NA/NULL** for marks drawn without them, so a plain scene still yields a geometry table.

Value

A list with two data frames:

- **elements** — one row per element: **key**, **mark**, **id**, **name**, **panel**, the device-px bbox **x0,y0,x1,y1**, its centre/size **x,y,w,h**, and a **meta** list-column.
- **panels** — one row per named panel: **name** and its elements' bounding box **x0,y0,x1,y1**.

See Also

[scene_svg\(\)](#)

scene_png

Render a scene to PNG or PDF bytes

Description

Like `render()` with a `.png` / `.pdf` path, but returns the encoded document as a raw vector instead of writing a file. These are the in-memory entry points for a host that needs the bytes rather than a path — embedding a base64 data URI in an HTML widget, serving a plot from a web API, or writing to a connection — without a temp-file round-trip.

Usage

```
scene_png(scene, scale = 1, cvd = "none")
```

```
scene_pdf(scene)
```

Arguments

scene	A <code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method.
scale	Resolution multiplier (default 1). <code>scale = 2</code> renders at twice the device pixels while keeping the same physical size — the retina / <code>ggsave(scoping=)</code> idiom. It multiplies <code>dpi</code> , so absolute units (<code>mm</code> , <code>pt</code> , <code>in</code>) cover proportionally more pixels and nothing about the layout changes; text does not get relatively bigger or smaller. Only raster output gains anything: a PDF's page size in points and an SVG's physical size are unchanged by construction.
cvd	Simulate a colour-vision deficiency: <code>"none"</code> (default), <code>"protanopia"</code> (red-blind), <code>"deutanopia"</code> (green-blind, the common one that breaks red/green encodings), <code>"tritanopia"</code> (blue-blind), or <code>"achromatopsia"</code> (total, and a fair proxy for greyscale printing). The simulation is a post-pass over the finished raster using the Machado et al. (2009) matrices applied in linear light, so it turns an accessibility check into a one-line change instead of an export-and-upload round trip. Raster output only — a vector format has no pixels to transform, and <code>render()</code> warns if you ask for one.

Details

Together with `scene_svg()` (a string) and `scene_raster()` (pixels), every output format vellum supports can now be produced without touching disk.

Backend degradation warnings (see `render()`) are surfaced here too, so a PDF that could not honour a pattern or mask reports it exactly as writing a file would.

Value

A raw vector: the encoded PNG or PDF document.

See Also

`render()`, `scene_svg()`, `scene_raster()`

Examples

```
s <- vl_scene(2, 2) |> draw(circle_grob(gp = vl_gpar(fill = "steelblue")))
png_bytes <- scene_png(s)
length(png_bytes)
rawToChar(png_bytes[2:4]) # "PNG"
```

scene_raster	<i>Read a rendered scene back as pixels</i>
--------------	---

Description

`scene_raster()` renders `scene` and returns its pixels as an integer array with dimensions `c(channel, x, y)` — RGBA channels in 0:255, top-left origin, `y` increasing downward. This is the form most convenient for probing or testing (e.g. `scene_raster(s)[1, x, y]` is the red value at pixel `(x, y)`).

Usage

```
scene_raster(scene, cvd = "none")
```

Arguments

scene	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method).
cvd	Simulate a colour-vision deficiency: "none" (default), "protanopia" (red-blind), "deutanopia" (green-blind, the common one that breaks red/green encodings), "tritanopia" (blue-blind), or "achromatopsia" (total, and a fair proxy for greyscale printing). The simulation is a post-pass over the finished raster using the Machado et al. (2009) matrices applied in linear light, so it turns an accessibility check into a one-line change instead of an export-and-upload round trip. Raster output only — a vector format has no pixels to transform, and <code>render()</code> warns if you ask for one.

Details

An `grDevices::as.raster()` method returns the same image as a `raster` object (a character matrix of hex colours), drawable with `graphics::plot()` or `grid::rasterGrob()`.

Value

`scene_raster()`: an integer array of dimension `c(4, width, height)`. The `as.raster()` method: a `raster` (character matrix, `c(height, width)`).

Examples

```
s <- vl_scene(2, 1, bg = "white") |>
  draw(circle_grob(r = 0.3, gp = vl_gpar(fill = "red", col = NA)))
dim(scene_raster(s)) # c(4, width_px, height_px)
```

scene_stats

Ink and overplotting statistics for a scene

Description

How much of the canvas a scene actually covers, and how hard it is working to do it. `overplot` is the honest signal for "should this be `datashade()`-ed?": a count of marks says nothing about whether they overlap, whereas thirty thousand well-separated points are fine and eight thousand piled on top of each other are not.

Usage

```
scene_stats(scene)
```

Arguments

`scene` A `vl_scene()`, or anything with an `as_vellum_scene()` method.

Value

A one-row data frame:

`elements` total drawn elements.

`ink` fraction of canvas pixels differing from the background.

`colours` distinct colours in the rendered image.

`overplot` summed element-box area divided by inked area — an estimate of how many times an average inked pixel was drawn over. 1 means no overlap. It is computed from bounding boxes, so it over-estimates for non-rectangular marks (a circle fills ~79% of its box); treat it as an index, not a measurement. Marks the element table does not cover — text, and unkeyed lines/polygons/paths — contribute to `ink` but not to `overplot`.

See Also

`vl_lint()`, `profile_render()`, `datashade()`

Examples

```
set.seed(1)
sparse <- vl_scene(4, 3) |> draw(points_grob(runif(200), runif(200)))
dense <- vl_scene(4, 3) |> draw(points_grob(rnorm(20000, 0.5, 0.05),
                                           rnorm(20000, 0.5, 0.05)))
rbind(sparse = scene_stats(sparse), dense = scene_stats(dense))
```

scene_svg	<i>Render a scene to an SVG string</i>
-----------	--

Description

Like `render()` with an `.svg` path, but returns the SVG document as a character string instead of writing a file. This is the in-memory entry point for hosting a scene interactively (an `htmlwidget` embeds the markup directly) and for tests that assert on emitted attributes such as `data-key`.

Usage

```
scene_svg(scene, text = c("native", "outline"))
```

Arguments

scene	A <code>vl_scene()</code> .
text	For SVG output, how text is written: "native" (default) emits selectable <code><text></code> referencing system fonts, "outline" emits glyph outlines (pixel-faithful, identical to the raster/PDF backends, but not selectable). Ignored for PNG/PDF.

Value

A length-1 character vector: the SVG document.

See Also

`render()`, `scene_png()`, `scene_model()`

scene_write	<i>Write and read a scene</i>
-------------	-------------------------------

Description

Persist a scene so it can be rebuilt later, in another session, or in another process. `.rds` is the default and needs nothing extra; `.json` produces a portable text format and needs the `jsonlite` package.

Usage

```
scene_write(scene, path)

scene_read(path)
```

Arguments

scene A `vl_scene()`, or anything with an `as_vellum_scene()` method.
path File path. The extension picks the format (`.rds` or `.json`).

Details

What this buys you: caching a built scene without re-running the code that built it, sending one over a wire for a worker to render, and keeping a reproducible intermediate between "the code" and "the pixels".

Value

`scene_write()`: path, invisibly. `scene_read()`: a `vellum_scene`.

See Also

`as_scene_spec()`, `scene_diff()`, `scene_hash()`

Examples

```
s <- vl_scene(2, 2) |> draw(circle_grob(gp = vl_gpar(fill = "steelblue")))
f <- tempfile(fileext = ".rds")
scene_write(s, f)
identical(as_scene_spec(scene_read(f)), as_scene_spec(s))
```

sketch	<i>Hand-drawn ("sketch") rendering</i>
--------	--

Description

Attach to a grob's `sketch` argument to render it in a hand-drawn, sketchy style (the **Rough.js** look): wobbly outlines and hachure fills. Supported by `rect_grob()`, `polygon_grob()`, `lines_grob()`, and `circle_grob()`. Output is deterministic given `seed`.

Usage

```
sketch(
  roughness = 1,
  bowing = 1,
  fill_style = c("hachure", "solid", "crosshatch", "zigzag", "dots"),
  fill_weight = NULL,
  hachure_angle = -41,
  hachure_gap = NULL,
  curve_tightness = 0,
  disable_multi_stroke = FALSE,
  preserve_vertices = FALSE,
  seed = 1L
)
```

Arguments

<code>roughness</code>	Wobble amount (≥ 0 ; 0 is nearly crisp, 1 the default hand-drawn look, higher is wilder).
<code>bowing</code>	How much straight edges bow (0 disables bowing).
<code>fill_style</code>	One of "hachure" (default), "solid", "crosshatch", "zigzag", "dots". Non-solid styles paint the fill colour as line work.
<code>fill_weight</code>	Stroke width of fill/hachure lines, in <code>lwd</code> units (1 == 1/96 inch); <code>NULL</code> derives it from the grob's <code>lwd</code> .
<code>hachure_angle</code>	Hachure line angle in degrees.
<code>hachure_gap</code>	Gap between hachure lines, in <code>lwd</code> units; <code>NULL</code> = auto.
<code>curve_tightness</code>	Curve fit tightness for round shapes (circles, arcs).
<code>disable_multi_stroke</code>	If <code>TRUE</code> , draw single (not doubled) outline strokes — a cleaner, less sketchy line.
<code>preserve_vertices</code>	If <code>TRUE</code> , keep shape vertices exact (only edges wobble).
<code>seed</code>	Integer seed for the wobble (same seed => identical output).

Details

Sketch is a deliberate exception to vellum's crisp, fidelity-first defaults — see `vignette / _docs/DESIGN-ROUGH.R`.md. Text is never sketched.

Value

A `vellum_sketch` object for a grob's `sketch` argument.

Examples

```
rect_grob(gp = vl_gpar(fill = "steelblue", col = "black"), sketch = sketch())
```

<code>spread</code>	<i>Spread (dilate) the pixels of a raster grob</i>
---------------------	--

Description

Grow each non-empty pixel of a raster over a small neighbourhood so thin marks stay visible. This is `datashader's spread`: useful after `datashade()`, `datashade_lines()`, or `datashade_segments()` when the output is sparse (single-pixel lines, isolated points). `dynspread()` chooses the radius automatically from the image's density; `spread()` uses a fixed one.

Usage

```
spread(grob, px = 1L, shape = c("circle", "square"), how = c("over", "add"))
```

Arguments

grob	A raster grob (e.g. from datashade()). Other grob kinds are returned unchanged.
px	Spread radius in pixels (non-negative integer). 0 returns grob unchanged.
shape	Neighbourhood shape: "circle" (default, Euclidean radius) or "square" (Chebyshev radius).
how	How overlapping spread pixels combine: "over" (default) keeps the colour of the most-opaque contributor; "add" accumulates opacity (clamped) so overlaps darken.

Value

A raster [grob](#) with the spread applied.

See Also

[dynspread\(\)](#), [datashade\(\)](#), [datashade_lines\(\)](#).

Examples

```
set.seed(1)
g <- datashade_segments(rnorm(2000), rnorm(2000), rnorm(2000), rnorm(2000))
g2 <- spread(g, px = 2)
```

stroke_to_path	<i>Freeze a stroke into a fillable outline</i>
-----------------------	--

Description

Converts the *stroke* of a line-like grob into a **path_grob** describing the region the stroke covers — the shape you would get by tracing round the drawn line. What was a one-pixel-wide path with a colour becomes an area with an interior, which is what you need in order to fill it with a gradient or a pattern, to send it to a cutting plotter or CNC tool, or to do geometry with it.

Usage

```
stroke_to_path(
  grob,
  width = 6,
  height = 4,
  dpi = 96,
  gp = NULL,
  lwd_profile = NULL,
  along = c("arclength", "vertex")
)
```

Arguments

<code>grob</code>	A <code>lines_grob()</code> , <code>polygon_grob()</code> , or <code>path_grob()</code> . Its coordinates must be resolvable without a viewport: <code>npc</code> , an absolute unit, or <code>native</code> (read against the root's default 0..1 scale, which is what a bare numeric means).
<code>width, height</code>	Page size in inches to resolve the geometry against.
<code>dpi</code>	Resolution to resolve the stroke width against.
<code>gp</code>	Optional <code>vl_gpar()</code> overriding the grob's own for the stroke parameters (<code>lwd</code> , <code>lineend</code> , <code>linejoin</code> , <code>linemitre</code>).
<code>lwd_profile</code>	Optional numeric vector of multipliers of <code>lwd</code> , describing how the width varies along the line: <code>c(1, 0.2)</code> tapers a ribbon to a fifth of its width, <code>c(0, 1, 0)</code> is a leaf, <code>c(1, 0, 1)</code> is pinched to nothing in the middle. Values must be finite and ≥ 0 , and at least one must be positive; a zero is a legal width, not an error. A single value simply scales the width. <code>NULL</code> (default) is a constant width, and takes exactly the expansion this function has always performed.
<code>along</code>	How <code>lwd_profile</code> is positioned. <code>"arclength"</code> (default) spreads its values evenly along the <i>drawn length</i> of the stroke and interpolates between them, so the profile is independent of how the polyline happens to be sampled and any profile length works. <code>"vertex"</code> reads one value per vertex, in point order, which is what you want when the widths come from data attached to the vertices — a per-observation weight, a pressure trace. The two coincide only when the vertices are evenly spaced.

Details

At constant width the expansion uses the same stroker the rasterizer uses, so the outline is exactly the region that would have been inked, not a reimplementaion that could drift from it. A *varying* width has no such reference — no rasterizer can stroke a line at a changing width — so `lwd_profile` is served by vellum's own offsetter instead. See **Variable width** below.

The result is baked at one size. A stroke width is a device quantity, so its outline only exists once a page size and resolution are chosen. Those are arguments here, and the returned coordinates are absolute (mm) — the outline will *not* rescale with the page the way the original stroke would. That is inherent: an outline is a shape, not a stroke.

Value

A `path_grob()` in mm units, with `rule = "winding"`, whose fill is the stroked region. Its `gp` starts from the source grob's stroke colour as a fill, so drawing it looks like the original line.

Variable width

The region swept by a round nib of varying radius is the union of the convex hulls of successive pairs of end discs, so `lwd_profile` builds that union rather than offsetting each

vertex along its normal. The distinction matters at a taper: an offset boundary would cut *into* the end disc and the tip would come out waisted rather than pointed.

Three consequences worth knowing:

- Width varies linearly **between vertices**, so a profile is only as smooth as the polyline under it. `bezier_grob()` and `spline_grob()` flatten to a polyline before this function sees them, so a profile rides the flattened points — prefer `along = "arclength"` there, and raise their `n` if a taper looks faceted.
- On a closed `polygon_grob()` or `path_grob()` the profile is **cyclic**: a ring has no first vertex a reader can see, so the values wrap. `c(1, 0)` on a ring is therefore two tapers, not one.
- `lwd_profile = NULL` is byte-identical to previous versions. A *constant* profile such as `lwd_profile = 1` is not — it goes through the offsetter, so it agrees with tiny-skia's stroker as a picture but not to the last bit. `lineend` and `linejoin` are honoured, but only on the outer silhouette; the inner side of a join is the natural overlap either way.

See Also

`path_grob()`, `grob`, `segments_grob()` for per-*element* widths

Examples

```
zig <- lines_grob(c(0.1, 0.35, 0.6, 0.9), c(0.2, 0.8, 0.2, 0.8),
                 gp = vl_gpar(col = "steelblue", lwd = 12))
outline <- stroke_to_path(zig, width = 3, height = 2)
# A width that varies along the line: thick in the middle, tapered at both ends.
leaf <- stroke_to_path(zig, width = 3, height = 2, lwd_profile = c(0.15, 1, 0.15))
# Now fillable: a gradient across the ribbon the line traced.
vl_scene(3, 2) |>
  draw(S7::set_props(outline, gp = vl_gpar(
    fill = linear_gradient(c("tomato", "gold")), col = "grey20", lwd = 0.5
  )))
```

style

Reusable style classes

Description

A **style** bundles `vl_gpar()` graphical parameters under an optional name so the same look can be reused across many viewports or grobs. A **style** is a **gpar** — it carries every graphical-parameter field and obeys the same inheritance rules — with an added **name** for identification. It can therefore be passed anywhere a **gp** is accepted.

Usage

```

style(
  col = NULL,
  fill = NULL,
  lwd = NULL,
  alpha = NULL,
  lty = NULL,
  lineend = NULL,
  linejoin = NULL,
  linemitre = NULL,
  fontfamily = NULL,
  fontface = NULL,
  fontsize = NULL,
  cex = NULL,
  lineheight = NULL,
  halo_col = NULL,
  halo_width = NULL,
  features = NULL,
  antialias = NULL,
  crisp = NULL,
  dash_phase = NULL,
  name = NULL
)

```

Arguments

<code>col</code>	Stroke/text colour, or a gradient from <code>linear_gradient()</code> / <code>radial_gradient()</code> to stroke with a gradient — the same paint model as <code>fill</code> , applied to the stroked region instead of the enclosed one. A gradient here affects stroked paths; text and markers fall back to its first stop, since a glyph run has no path to run a ramp along.
<code>fill</code>	Fill colour, or a gradient from <code>linear_gradient()</code> / <code>radial_gradient()</code> .
<code>lwd</code>	Line width (1 == 1/96 inch).
<code>alpha</code>	Opacity multiplier in [0, 1].
<code>lty</code>	Line type: a name ("solid", "dashed", "dotted", "dotdash", "longdash", "twodash"), an integer code 0:6, a hex dash string (e.g. "44"), or a numeric vector of on/off dash lengths. Dash lengths scale with <code>lwd</code> .
<code>lineend</code>	Line cap: "round" (default), "butt", or "square".
<code>linejoin</code>	Line join: "round" (default), "mitre", or "bevel".
<code>linemitre</code>	Mitre limit (≥ 1) for mitre joins; default 10.
<code>fontfamily</code>	Font family (text grobs).
<code>fontface</code>	One of "plain", "bold", "italic", "bold.italic".
<code>fontsize</code>	Font size in points.
<code>cex</code>	Multiplier applied to <code>fontsize</code> (grid semantics), so a theme can ask for a relative size without knowing the base one. <code>cex = 2</code> is exactly

	equivalent to doubling <code>fontsize</code> : it scales drawn text, <code>char/line</code> units, and <code>grobwidth()/grobheight()</code> measurement alike. NULL (the default) means 1.
<code>lineheight</code>	Line-height multiple.
<code>halo_col</code>	Halo colour for text (a "shadowtext" outline drawn <i>under</i> the glyphs, so a label stays legible over dense marks or map imagery). NULL (default) means no halo. Needs <code>halo_width</code> to be visible.
<code>halo_width</code>	Halo thickness in points – the visible width outside the glyph. NULL or 0 means no halo. A good starting point is roughly an eighth of <code>fontsize</code> .
<code>features</code>	OpenType font features, as a named numeric vector of four-character feature tags – e.g. <code>c(tnum = 1)</code> for tabular (fixed-width) figures so axis labels stop jittering between ticks, <code>c(smcp = 1)</code> for small caps, <code>c(onum = 1)</code> for oldstyle figures, <code>c(liga = 0)</code> to switch ligatures off, or <code>c(kern = 0)</code> to disable kerning. NULL (default) uses the font's own defaults. A feature the font does not carry is silently ignored – that is HarfBuzz's behaviour, not something vellum can check for you.
<code>antialias</code>	Anti-alias this element's edges? NULL (default) inherits; the root default is TRUE. FALSE gives hard pixel edges, which is what pixel art, QR codes, and heatmap cells that must tile without a seam want.
<code>crisp</code>	Snap axis-parallel strokes onto the pixel grid? NULL (default) inherits; the root default is FALSE. A 1-px rule at a fractional coordinate straddles two pixel rows and renders as two grey ones rather than one solid — the reason gridlines look muddy on screen. TRUE snaps horizontal and vertical runs so they land on whole pixels. Diagonals are unaffected (there is no grid to snap them to), and it only applies to raster output — a vector format has no pixel grid.
<code>dash_phase</code>	How far into the dash pattern a dashed line starts, as a multiple of <code>lwd</code> (so it scales with the line width exactly as the dash nibbles do). Use it to line up dashes across adjacent strokes, or animate it for marching ants. NULL (default) means 0. Ignored for a solid line.
<code>name</code>	Optional style-class name, for identification only; it is ignored by rendering.

Details

Attaching a `style` to a viewport cascades its defaults to the whole subtree via the ordinary gpar inheritance (more-specific overrides less-specific), so a child grob's own `gp` still wins. This is the reusable "style class" layer that sits below a grammar's themes: a theme can compile *into* named styles rather than setting gpar fields ad hoc on every element.

Value

A `style` object (a subclass of `gpar`).

Examples

```
accent <- style(col = "firebrick", lwd = 2, name = "accent")
```

```
# Reuse it on a viewport; children inherit unless they override.
vl_viewport(gp = accent)
```

text_path_grob	<i>Text set along a path</i>
----------------	------------------------------

Description

`text_path_grob()` places a label along a polyline baseline instead of at a point: each glyph keeps the pen position shaping gave it and is drawn at that distance along the path, rotated to the local tangent. Use it for labels that follow an arc, a contour, a river or a curved axis.

Usage

```
text_path_grob(
  label,
  x,
  y,
  just = "centre",
  offset = 0,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)
```

Arguments

<code>label</code>	A single character string.
<code>x, y</code>	The baseline path, as unit vectors of equal length.
<code>just</code>	Justification: <code>c(hjust, vjust)</code> as names ("left", "centre", "right", "bottom", "top") or numbers in [0, 1].
<code>offset</code>	Perpendicular standoff from the baseline in points; positive is to the left of the direction of travel.
<code>gp</code>	Graphical parameters, from <code>vl_gpar()</code> .
<code>name</code>	Optional name (for <code>edit_node()</code>).
<code>vp</code>	Optional <code>vl_viewport()</code> to draw this grob inside.
<code>id</code>	For most grobs, an optional semantic identifier emitted by the SVG backend as <code>data-vellum-id</code> (for interactivity, accessibility, and testing; ignored by raster/PDF). For path_grob only , <code>id</code> instead groups points (one value per point) into closed sub-paths: all points sharing an <code>id</code> form one sub-path (so a hole is a separate <code>id</code>), in first-appearance order (à la grid); <code>NULL</code> makes a single sub-path.
<code>role</code>	Optional ARIA role, emitted by the SVG backend as <code>role=</code> for accessibility (ignored by the raster and PDF backends).

Details

The path is a sequence of points, exactly like `lines_grob()` — pass the output of `bezier_grob()`-style control points already flattened, or any polyline. Arc length is measured on the *rendered* path, so the result adapts to the viewport the grob is drawn in.

`just` does double duty: the horizontal component slides the run along the baseline ("`left`" starts at the first point, "`centre`" centres it, "`right`" ends at the last), and the vertical component positions the glyphs against the baseline as it would for ordinary text. For finer control of the standoff — a label riding just above a curve rather than sitting on it — use `offset`.

Glyphs follow the tangent, as in SVG `textPath`: a label on the underside of a closed curve reads upside-down. That is the honest result of the geometry, and the fix is to reverse the path rather than the glyphs — set the lower arc as a second `text_path_grob()` whose points run the other way.

Glyphs are placed one per character. For Latin text that is exact; scripts where shaping produces a different number of glyphs than characters (Arabic, Devanagari, and ligature-heavy fonts) will still draw, but the per-glyph text recovered by PDF copy-paste and by SVG native text degrades to the whole label on the first glyph.

Value

A grob.

Examples

```
th <- seq(pi, 0, length.out = 60)
vl_scene(4, 2.2, dpi = 96, bg = "white") |>
  draw(text_path_grob(
    "text that follows a curve",
    x = 0.5 + 0.42 * cos(th), y = 0.15 + 0.7 * sin(th),
    offset = 3, gp = vl_gpar(fontsize = 13)
  ))
```

vl_arrow

Arrowheads

Description

Describe arrowheads to draw on the ends of a `lines_grob()` or `segments_grob()` (pass as their `arrow = argument`).

Usage

```
vl_arrow(
  angle = 30,
  length = vl_unit(0.25, "in"),
  ends = c("last", "first", "both"),
  type = c("open", "closed")
)
```

Arguments

angle	Half-angle of the head at the tip, in degrees (default 30).
length	Head length as an absolute <code>vl_unit()</code> (default <code>vl_unit(0.25, "in")</code>).
ends	Which ends get a head: "last" (default), "first", or "both".
type	"open" (a two-barb V) or "closed" (a filled triangle).

Value

A `vellum_arrow` object.

Examples

```
lines_grob(c(0.1, 0.9), c(0.1, 0.9), arrow = vl_arrow(type = "closed"))
```

<code>vl_buffer</code>	<i>Buffer a polygon outward</i>
------------------------	---------------------------------

Description

Grows a ring by **width** in every direction — the exclusion zone around a region, the halo around a cluster outline, the standoff a leader line should keep.

Usage

```
vl_buffer(x, y, width, arc = 6)
```

Arguments

x, y	The ring's coordinates, in order and not closed.
width	Distance to grow by, in the same units as <code>x/y</code> .
arc	Points per quarter-turn on rounded corners. More is smoother.

Details

Corners are rounded, which is what a standoff usually wants and what keeps the result well-defined at sharp angles (a mitred corner runs away to infinity as the angle closes).

On concave rings: the offset boundary of a concave polygon can cross itself where the buffer is wider than a local feature. This function offsets the ring and does not repair that, because repairing it is a boolean union — a Phase 12 operation. Buffering a convex ring, or buffering by less than the narrowest feature, is always well-behaved. Passing the result through `vl_hull()` is a crude but effective fix when it is not.

Value

A data frame of the buffered ring's `x/y`.

Examples

```
tri <- data.frame(x = c(0.3, 0.7, 0.5), y = c(0.3, 0.35, 0.7))
buf <- vl_buffer(tri$x, tri$y, 0.08)
vl_scene(3, 3, dpi = 96, bg = "white") |>
  draw(polygon_grob(buf$x, buf$y, gp = vl_gpar(fill = "#F3E0DA", col = NA))) |>
  draw(polygon_grob(tri$x, tri$y, gp = vl_gpar(fill = "tomato", col = NA)))
```

`vl_clear_render_cache`

Clear the render cache

Description

vellum memoises compiled scenes keyed on an object-identity token so repeat renders of an unchanged scene (multi-format export, a `display()` resize back to a prior size, or animation replaying a fixed set of frames) are cheap. The cache is transparent — a cached render is byte-identical to an uncached one — and bounded (`options(vellum.cache_size=)`, default 8), so you rarely need this; it is provided to reclaim memory or to force a cold render in benchmarks. Disable caching entirely with `options(vellum.cache = FALSE)`.

Usage

```
vl_clear_render_cache()
```

Value

NULL, invisibly.

Examples

```
vl_clear_render_cache()
```

`vl_contour`

Contour lines from a grid

Description

Marching squares over a matrix, chained into polylines. Use it for density isolines over a [datashade\(\)](#) surface, level curves of a fitted model, or any other gridded field.

Usage

```
vl_contour(z, levels = NULL, xlim = c(0, 1), ylim = c(0, 1))

contour_grob(
  contours,
  close = TRUE,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  role = NULL
)
```

Arguments

z	A numeric matrix, with rows indexing x and columns indexing y — <code>dim(z) == c(length(x), length(y))</code> , the same convention as <code>graphics::image()</code> , <code>graphics::contour()</code> and <code>graphics::persp()</code> , and the shape <code>outer(xs, ys, f)</code> produces.
levels	Contour levels. Defaults to 5 levels spanning the finite range of z , excluding the extremes (a contour at the minimum or maximum is either empty or the whole boundary).
xlim, ylim	The coordinate range the grid spans. Cell <i>centres</i> are placed at the ends of these ranges, matching <code>image()</code> .
contours	The data frame returned by <code>vl_contour()</code> .
close	Draw closed contours as closed rings. <code>TRUE</code> (the default) joins the last point back to the first for contours <code>vl_contour()</code> marked <code>closed</code> ; open contours are never closed.
gp, name, vp, role	Passed to each <code>lines_grob()</code> the contours become.

Details

Segments are chained into continuous lines rather than returned loose. That matters beyond tidiness: an unchained contour restarts its dash pattern in every grid cell, cannot be simplified or measured, and cannot be filled. Closed rings come back closed.

Cells with a non-finite corner are skipped, so a contour **breaks** around missing data rather than being drawn through it.

Value

A data frame with one row per point: `level`, `id` (a distinct contour line), `x`, `y`, and `closed`. Draw it with `lines_grob(x, y, id = id)`.

Examples

```
# A ring contour around a Gaussian bump.
# rows index x, columns index y -- exactly what `outer(xs, ys, f)` gives.
g <- outer(seq(-3, 3, length.out = 60), seq(-3, 3, length.out = 60),
```

```

      function(x, y) exp(-(x^2 + y^2) / 2))
head(vl_contour(g, levels = c(0.2, 0.5, 0.8)))
# Draw them: one grob per contour, because each is its own polyline.
# rows index x, columns index y -- exactly what `outer(xs, ys, f)` gives.
g <- outer(seq(-3, 3, length.out = 60), seq(-3, 3, length.out = 60),
           function(x, y) exp(-(x^2 + y^2) / 2))
vl_scene(3, 3, dpi = 96, bg = "white") |>
  push(vl_viewport(xscale = c(-3, 3), yscale = c(-3, 3))) |>
  draw(contour_grob(vl_contour(g, levels = c(0.2, 0.5, 0.8)),
                   gp = vl_gpar(col = "steelblue"))) |>
  pop() |>
  (\(s) s)()

```

vl_convert

Convert a unit to another unit, in a scene's context

Description

The counterpart to grid's `convertWidth()/convertHeight()/convertX()/convertY()`: resolve a `vl_unit()` vector to a plain number in some other unit. `why_size()` explains a *named node's* resolved size and `scene_model()` reports resolved boxes for *keyed elements*; this answers the remaining question — “how big is this particular unit, here?” — which a layer built on vellum needs whenever it has to size something itself.

Usage

```

vl_convert(
  u,
  to = "mm",
  scene = NULL,
  name = NULL,
  axis = c("x", "y"),
  what = c("length", "position")
)

```

Arguments

u	A <code>vl_unit()</code> vector (or a bare numeric, read as <code>npc</code>).
to	Target unit: "mm" (default), "cm", "in", "pt", "px", "npc", or "native".
scene	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method) giving the context. Required unless every input <i>and</i> <code>to</code> is absolute.
name	Name of the viewport to resolve against. NULL (default) uses the whole page.
axis	Which extent relative units refer to: "x" (default) or "y".
what	"length" (default) for a size, "position" for a coordinate. Only affects <code>native</code> .

Details

Absolute units (`mm`, `cm`, `in`, `pt`) convert with no context, so `scene` may be omitted. Relative units (`npc`, `native`) need to know the region they are relative to, and therefore need a `scene` — and, for anything other than the whole page, the `name` of a viewport in it.

Where `grid` uses four functions, this uses two arguments: `axis` picks the x or y extent, and `what` distinguishes a **length** from a **position**. They differ only for `native`, and only when the scale does not start at zero: on `xscale = c(10, 20)`, `unit(12, "native")` is *two tenths* of the width as a position, but *twelve tenths* as a length.

Value

A numeric vector the same length as `u`, in `to` units.

See Also

`why_size()`, `scene_model()`, `vl_unit()`

Examples

```
s <- vl_scene(4, 3, dpi = 100)
vl_convert(vl_unit(1, "in"), "mm") # absolute: no scene needed
vl_convert(vl_unit(0.5, "npc"), "mm", s) # half the page width
vl_convert(vl_unit(0.5, "npc"), "mm", s, axis = "y")

# A named viewport, and the length/position distinction on a shifted scale.
s2 <- s |> push(vl_viewport(width = 0.5, xscale = c(10, 20), name = "panel"))
vl_convert(vl_unit(12, "native"), "mm", s2, name = "panel")
vl_convert(vl_unit(12, "native"), "mm", s2, name = "panel", what = "position")
```

`vl_empty_region`

Find the largest empty rectangle in a scene

Description

Answers “where is there room?” — for a legend, an annotation, a watermark, or anything else that has to go somewhere the marks are not.

Usage

```
vl_empty_region(
  scene,
  avoid = NULL,
  within = NULL,
  grid = 200,
  unit = c("npc", "mm")
)
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method).
<code>avoid</code>	Names of the grobs to treat as obstacles. NULL (default) takes every node that is not a label.
<code>within</code>	Region to search, as <code>c(x0, y0, x1, y1)</code> in <code>npc</code> of the page. Defaults to the whole page.
<code>grid</code>	Grid resolution (cells across the longer side).
<code>unit</code>	Unit of the returned rectangle: <code>"npc"</code> (default) or <code>"mm"</code> .

Details

Occupancy is rasterised onto a grid and the answer is exact *on that grid*. The approximation is deliberate: the exact maximal empty rectangle over n boxes is superquadratic, and nothing is placed to sub-pixel tolerance. Raise `grid` to trade time for precision. Boxes are rounded outward, so the result is conservative — it will never claim space that is in fact occupied.

Value

A named numeric `c(x0, y0, x1, y1)`, in `unit`, using the usual y-grows-up convention. All zeros if there is no free space.

Examples

```
set.seed(1)
s <- vl_scene(4, 3, dpi = 96, bg = "white") |>
  draw(points_grob(runif(40) * 0.5, runif(40),
    gp = vl_gpar(fill = "steelblue", col = NA)))
vl_empty_region(s)
```

<code>vl_gpar</code>	<i>Graphical parameters</i>
----------------------	-----------------------------

Description

Builds a set of graphical parameters attached to a grob or viewport. Any field left NULL is inherited from the enclosing viewport; `alpha` multiplies down the viewport tree. A colour value sets it; NA means "no paint".

Usage

```
vl_gpar(
  col = NULL,
  fill = NULL,
  lwd = NULL,
  alpha = NULL,
  lty = NULL,
```

```

    lineend = NULL,
    linejoin = NULL,
    linemitre = NULL,
    fontfamily = NULL,
    fontface = NULL,
    fontsize = NULL,
    cex = NULL,
    lineheight = NULL,
    halo_col = NULL,
    halo_width = NULL,
    features = NULL,
    antialias = NULL,
    crisp = NULL,
    dash_phase = NULL
)

```

Arguments

<code>col</code>	Stroke/text colour, or a gradient from <code>linear_gradient()</code> / <code>radial_gradient()</code> to stroke with a gradient — the same paint model as <code>fill</code> , applied to the stroked region instead of the enclosed one. A gradient here affects stroked paths; text and markers fall back to its first stop, since a glyph run has no path to run a ramp along.
<code>fill</code>	Fill colour, or a gradient from <code>linear_gradient()</code> / <code>radial_gradient()</code> .
<code>lwd</code>	Line width (1 == 1/96 inch).
<code>alpha</code>	Opacity multiplier in [0, 1].
<code>lty</code>	Line type: a name ("solid", "dashed", "dotted", "dotdash", "longdash", "twodash"), an integer code 0:6, a hex dash string (e.g. "44"), or a numeric vector of on/off dash lengths. Dash lengths scale with <code>lwd</code> .
<code>lineend</code>	Line cap: "round" (default), "butt", or "square".
<code>linejoin</code>	Line join: "round" (default), "mitre", or "bevel".
<code>linemitre</code>	Mitre limit (≥ 1) for mitre joins; default 10.
<code>fontfamily</code>	Font family (text grobs).
<code>fontface</code>	One of "plain", "bold", "italic", "bold.italic".
<code>fontsize</code>	Font size in points.
<code>cex</code>	Multiplier applied to <code>fontsize</code> (grid semantics), so a theme can ask for a relative size without knowing the base one. <code>cex = 2</code> is exactly equivalent to doubling <code>fontsize</code> : it scales drawn text, <code>char/line</code> units, and <code>grobwidth()/grobheight()</code> measurement alike. NULL (the default) means 1.
<code>lineheight</code>	Line-height multiple.
<code>halo_col</code>	Halo colour for text (a "shadowtext" outline drawn <i>under</i> the glyphs, so a label stays legible over dense marks or map imagery). NULL (default) means no halo. Needs <code>halo_width</code> to be visible.

halo_width	Halo thickness in points – the visible width outside the glyph. <code>NULL</code> or <code>0</code> means no halo. A good starting point is roughly an eighth of <code>fontsize</code> .
features	OpenType font features, as a named numeric vector of four-character feature tags – e.g. <code>c(tnum = 1)</code> for tabular (fixed-width) figures so axis labels stop jittering between ticks, <code>c(smcp = 1)</code> for small caps, <code>c(onum = 1)</code> for oldstyle figures, <code>c(liga = 0)</code> to switch ligatures off, or <code>c(kern = 0)</code> to disable kerning. <code>NULL</code> (default) uses the font’s own defaults. A feature the font does not carry is silently ignored – that is HarfBuzz’s behaviour, not something vellum can check for you.
antialias	Anti-alias this element’s edges? <code>NULL</code> (default) inherits; the root default is <code>TRUE</code> . <code>FALSE</code> gives hard pixel edges, which is what pixel art, QR codes, and heatmap cells that must tile without a seam want.
crisp	Snap axis-parallel strokes onto the pixel grid? <code>NULL</code> (default) inherits; the root default is <code>FALSE</code> . A 1-px rule at a fractional coordinate straddles two pixel rows and renders as two grey ones rather than one solid — the reason gridlines look muddy on screen. <code>TRUE</code> snaps horizontal and vertical runs so they land on whole pixels. Diagonals are unaffected (there is no grid to snap them to), and it only applies to raster output — a vector format has no pixel grid.
dash_phase	How far into the dash pattern a dashed line starts, as a multiple of <code>lwd</code> (so it scales with the line width exactly as the dash nibbles do). Use it to line up dashes across adjacent strokes, or animate it for marching ants. <code>NULL</code> (default) means 0. Ignored for a solid line.

Value

A `gpar` object.

Examples

```
vl_gpar(col = "steelblue", lwd = 2, lty = "dashed", lineend = "round")
```

`vl_hatch`

Hatch fill

Description

Fills a shape with ruled parallel lines. Unlike `vl_pattern()`, which rasterises a tile, a hatch is **geometry**: it stays crisp at any zoom, prints correctly, and survives being converted to greyscale.

Usage

```
vl_hatch(angle = 45, spacing = 3, width = 0.75, col = "black", bg = NA)
```

Arguments

angle	Direction of the rules, in degrees counter-clockwise from horizontal. Distinct angles (0, 45, 90, 135) read as distinct categories.
spacing	Distance between rules, in points.
width	Rule width, in points.
col	Rule colour.
bg	Optional background painted behind the rules. NA (default) leaves whatever is underneath showing through.

Details

That last point is the reason to reach for it. A colour encoding that fails for a red/green-blind reader — which `render()`'s `cvd` argument will show you and `vl_lint()` will flag — is fixed by encoding with *texture* as well as hue. Hatching is the standard way to do that.

Value

A `vellum_hatch` object, usable anywhere a `fill` is.

See Also

`vl_pattern()` for a tiled grob, `linear_gradient()`, `vl_lint()`

Examples

```
vl_scene(3, 2) |>
  draw(rect_grob(width = 0.8, height = 0.8, gp = vl_gpar(
    fill = vl_hatch(angle = 45, spacing = 4), col = "grey30"
  )))
```

vl_hull
Hull of a point set

Description

`vl_hull()` returns the convex hull, or a concave one that follows the point set more closely — useful for outlining a cluster, or for building an exclusion zone that annotations should avoid.

Usage

```
vl_hull(x, y, concavity = Inf)
```

Arguments

x, y	Point coordinates (plain numerics, in whatever space you are working in — the hull is scale-free).
concavity	Threshold as above; larger is more convex, and Inf (the default) is the convex hull.

Details

The concave hull digs into the convex hull, replacing an edge with two edges through the nearest unused point whenever the edge is long relative to the detour. **concavity** is that threshold, and **larger means more convex**:

- **Inf** (the default) is exactly the convex hull.
- **8** follows the points loosely.
- **4** is a good tight outline for a scattered cluster.
- Below about **3** the boundary starts to self-intersect.

Self-intersection is inherent to the method, not a defect to be fixed here — a boundary that threads between interior points has to cross itself eventually. Inspect the result if you push it hard.

Value

A data frame of the hull ring's x/y, in order, ready for `polygon_grob()`. The ring is not closed (the last point does not repeat the first).

Examples

```
set.seed(1)
pts <- data.frame(x = runif(40), y = runif(40))
h <- vl_hull(pts$x, pts$y)
vl_scene(3, 3, dpi = 96, bg = "white") |>
  draw(polygon_grob(h$x, h$y, gp = vl_gpar(fill = "#DCE7F5", col = "steelblue"))) |>
  draw(points_grob(pts$x, pts$y, gp = vl_gpar(fill = "grey25", col = NA)))
```

vl_lint

Check a scene for likely mistakes

Description

Static analysis for graphics. `vl_lint()` resolves the scene and inspects the geometry the renderer would draw with, reporting the things people normally find by squinting at the output: a mark that landed off-canvas, a label too small to read, text that will not contrast with what is behind it, an element that is invisible because nothing was ever going to paint it.

Usage

```
vl_lint(
  scene,
  rules = NULL,
  exclude = NULL,
  severity = NULL,
  min_text_px = 7,
```

```

    min_text_pt = 6,
    min_contrast = 3,
    max_overplot = 8,
    cvd = "deuteranopia",
    min_cvd_delta = 0.06
)

```

Arguments

scene	A <code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method.
rules	Rule ids to run; NULL (default) runs all registered rules.
exclude	Node names (or ids, or kinds) whose findings to drop — the deliberate oddity you have already looked at and accepted. Suppression is by node because rules already selects rules, and without it a figure with one intentional off-canvas mark could never reach a clean lint to assert on. An entry matching nothing in the scene warns, since a stale exclude list looks exactly like a working one.
severity	Named character vector overriding a rule's own severity, e.g. <code>c(tiny_text = "note")</code> . Useful when a project cares about a rule more, or less, than vellum does.
min_text_px	Text shorter than this many device pixels is flagged as illegible. Default 7.
min_text_pt	Text smaller than this many points is flagged as illegible. Default 6. The two floors answer different questions and tiny_text fires on either. Device pixels decide whether the glyphs survive rasterisation, and points decide whether a human can read them — which is why a pixel floor alone is not enough: font_px scales with dpi , so a 4 pt label rendered at dpi = 300 clears a 7 px floor comfortably while remaining illegible on paper.
min_contrast	Minimum text-to-backdrop contrast ratio before low_contrast fires. Default 3 (WCAG AA for large text); WCAG AA for body text is 4.5.
max_overplot	How many times an average pixel inside a batched mark's own extent may be painted before overplotted fires. Default 8; a well-spread scatter sits below 1, a few thousand points around 4, and a dense cluster in the hundreds.
cvd	Colour-vision deficiencies cvd_collision should check: "deuteranopia" (the default, and the most common form), "protanopia", "tritanopia", "achromatopsia" — which doubles as "will this survive a greyscale printer" — or "none" to switch the rule off. Simulated with the same matrices <code>render(cvd =)</code> draws with.
min_cvd_delta	How close two colours may come, as an Oklab distance after simulation, before cvd_collision calls them the same colour. Default 0.06. Two references for that number: ggplot2's default three-colour palette puts red and green 0.325 apart in normal vision and 0.048 apart under deuteranopia, while the closest pair in the CVD-safe Okabe-Ito palette stays at 0.075.

Details

This is possible because vellum resolves layout and text metrics *before* drawing. A layer above grid cannot ask "how many pixels tall is this label", because the answer does not exist until a device is open.

The rule set is extensible — see `vl_lint_rule()`.

Value

A data frame of findings, empty if the scene is clean: `rule`, `severity`, `node`, `message`, and the finding's device-px box (`x0`, `y0`, `x1`, `y1`, `NA` where a rule reported something with no geometry). Printing shows a grouped summary.

A rule that fails does not abort the lint: it is reported as a `rule_error` finding naming the rule, and the other rules still run.

See Also

`vl_lint_rule()`, `scene_stats()`, `why_size()`

Examples

```
# A label pushed off the page, and one too small to read.
s <- vl_scene(3, 2) |>
  draw(text_grob("off the edge", x = 1.6, y = 0.5)) |>
  draw(text_grob("tiny", x = 0.5, y = 0.2, gp = vl_gpar(fontsize = 2)))
vl_lint(s)
```

<code>vl_lint_assert</code>	<i>Fail on lint findings</i>
-----------------------------	------------------------------

Description

`vl_lint()` for a test suite or a CI job: run the rules and stop if anything turns up. A figure that lints clean today stays that way, in the same spirit as `font_pin()` and `scene_hash()` — with the difference that this one catches defects rather than changes.

Usage

```
vl_lint_assert(
  scene,
  ...,
  severity = c("warning", "note"),
  on = c("error", "warn")
)
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method.
<code>...</code>	Passed to <code>vl_lint()</code> — <code>rules</code> , <code>exclude</code> , the thresholds.
<code>severity</code>	"warning" (default) fails only on warnings; "note" fails on anything at all.
<code>on</code>	"error" (default) or "warn".

Value

Invisibly, the findings that triggered the failure — a zero-row frame when the scene is clean.

See Also

`vl_lint()`, `vl_lint_overlay()` to see what was reported.

Examples

```
clean <- vl_scene(3, 2) |> draw(text_grob("fine", gp = vl_gpar(fontsize = 12)))
vl_lint_assert(clean)

# In a test:
# test_that("the figure has no lint findings", vl_lint_assert(my_plot()))
```

<code>vl_lint_finding</code>	<i>Build a lint finding</i>
------------------------------	-----------------------------

Description

The return shape `vl_lint_rule()` functions must produce. Vectorised over `nodes`, so a rule matches rows and describes them in one call.

Usage

```
vl_lint_finding(rule, severity = c("warning", "note"), nodes, message)
```

Arguments

<code>rule</code>	Rule id.
<code>severity</code>	"warning" (likely a real defect) or "note".
<code>nodes</code>	The matching rows of the node table.
<code>message</code>	One message, or one per row.

Value

A data frame of findings, one row per node, with the node's device-px box carried through in `x0/y0/x1/y1` so a caller can point at the finding on the image. The box is `NA` when `nodes` has no geometry.

vl_lint_overlay	<i>Draw lint findings onto the scene</i>
-----------------	--

Description

Puts a box around every finding that has geometry, and labels it with the rules that fired. For a graphics linter this is usually the fastest way to understand a report: a message says a mark is clipped, an outline shows you which one.

Usage

```
vl_lint_overlay(scene, findings = NULL, labels = TRUE)
```

Arguments

scene	A <code>vl_scene()</code> , or anything with an <code>as_vellum_scene()</code> method.
findings	The result of <code>vl_lint()</code> . NULL (default) lints <code>scene</code> .
labels	Whether to write the rule name beside each box.

Details

Findings with no geometry (a `rule_error`, or a rule reporting something scene-wide) have nothing to point at and are skipped.

Value

A new scene: `scene` with the overlay drawn on top, at page level.

See Also

`vl_lint()`, `vl_lint_assert()`

Examples

```
s <- vl_scene(3, 2) |>
  draw(text_grob("off the edge", x = 1.6, y = 0.5)) |>
  draw(text_grob("tiny", x = 0.5, y = 0.2, gp = vl_gpar(fontsize = 2)))
# display(vl_lint_overlay(s))
nrow(vl_lint(s)) > 0
```

vl_lint_rule	<i>Register a lint rule</i>
--------------	-----------------------------

Description

Adds a rule to the registry `vl_lint()` runs. A package layered on vellum can register rules of its own — vellum supplies the geometric ones (a mark drawn off-canvas, text too small to read), a grammar layer can add semantic ones (a scale with a single level, a legend with forty entries) — and both come back from one `vl_lint()` call.

Usage

```
vl_lint_rule(
  name,
  fn,
  description = "",
  kinds = NULL,
  needs_pixels = FALSE,
  tags = character()
)

vl_lint_rules()
```

Arguments

name	Short rule id, e.g. "tiny_text". Re-registering replaces.
fn	A function of (<code>scene</code>, <code>nodes</code>, <code>ctx</code>) returning a data frame of findings — build it with <code>vl_lint_finding()</code> — or <code>NULL</code> for "nothing found". A rule that fails is reported as a <code>rule_error</code> finding rather than aborting the lint. <code>nodes</code> is the resolved per-node table, one row per drawn node in paint order: <code>kind</code>, <code>name</code>, <code>id</code>, <code>label</code> what the node is, and what it was called. <code>label</code> is the string, for text nodes. <code>node</code> the paint index — later means drawn on top. <code>x0</code>, <code>y0</code>, <code>x1</code>, <code>y1</code> the device-px box, y-down. For a batched mark this is the union over every element, so reach for <code>ctx\$elements()</code> when that distinction matters. <code>clip_x0...clip_y1</code> the innermost clip box, which is the whole page when the node is unclipped. <code>vp</code>, <code>vp_x0...vp_y1</code> the node's viewport: an id that nodes sharing a viewport share, and that viewport's own device-px extent. Not the same as the clip box — an unclipped viewport still has an extent, and a mark can leave it. <code>n</code> how many elements the node draws.

	alpha, has_fill, has_col the group alpha, and whether a fill and a stroke are present at all. col, fill stroke and fill colour as 0xRRGGBBAA packed into a double (not an integer — the value overflows a signed 32-bit int once red reaches 128). Group alpha is already folded in. fill_kind "none", "solid", "linear", "radial", "pattern" or "hatch". A gradient has no single colour, so a rule reasoning about colour must check this before reading fill. lwd_px stroke width in device pixels. font_px text size in device pixels; 0 for everything else. notdef how many characters of a text node shaped to glyph 0 — no font on this machine has them, and they will render as tofu boxes. ctx carries: w, h, dpi the page in device pixels, and its resolution. min_text_px, min_text_pt, min_contrast, max_overplot, cvd, min_cvd_delta the thresholds vl_lint() was called with. pixel(x, y) one composited RGBA pixel, as a length-4 vector. region(x0, y0, x1, y1) every composited pixel in a box, as a 4-column RGBA matrix — cheaper and steadier than probing point by point, since a single probe can land on an incidental gridline. elements() the per-element table (key, panel, name, kind, node, box), which is what to use when a node is a batch: a scatter is one node whose box is the union over every point. pixel(), region() and elements() are all lazy. Rendering the scene to look at it is the expensive part of linting, and most rules never need to.
description	One line, shown by vl_lint_rules() .
kinds	Node kinds the rule looks at, e.g. "text". The rule is skipped when the scene contains none of them. NULL (default) means "any".
needs_pixels	Whether the rule renders the scene. Reported by vl_lint_rules() so a caller can select the cheap rules for a tight loop.
tags	Free-form labels for grouping rules, e.g. "accessibility".

Value

Invisibly, **name**.

vl_lint_rules(): a data frame of registered rules, with the **kinds** a rule looks at, whether it renders the scene, and its **tags**.

See Also

[vl_lint\(\)](#), [vl_lint_rules\(\)](#)

Examples

```
vl_lint_rule("no_hexagons", function(scene, nodes, ctx) {
  hits <- nodes[nodes$kind == "hexagon", , drop = FALSE]
  if (!nrow(hits)) return(NULL)
  vl_lint_finding("no_hexagons", "note", hits, "hexagons are banned here")
}, "example rule", kinds = "hexagon")
```

vl_nearest

Find the marks nearest a point, by true geometry

Description

Ranks a scene's keyed elements by their distance to a point — measuring to the **actual geometry**, not to a bounding box.

Usage

```
vl_nearest(scene, x, y, units = c("npc", "px"), n = 1L, max_dist = Inf)
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method).
<code>x, y</code>	The probe point.
<code>units</code>	"npc" (default, y up) or "px" (device pixels, y down) — the same convention as <code>hit_test()</code> .
<code>n</code>	How many nearest elements to return.
<code>max_dist</code>	Ignore anything further than this many device pixels.

Value

A data frame of `key`, `kind` and `dist` (device px), nearest first. Zero rows if nothing is in range. Only *keyed* elements are considered — an unkeyed mark is not addressable, so reporting it would be no use.

Why not the bounding box

For a round or upright mark the two agree. For anything diagonal or thin they do not, and the difference is not subtle: a line from the bottom-left of a panel to the top-right has a bounding box covering the *entire panel*, so a box-based "what is nearest" reports it from anywhere in the plot. That is why a host holding only boxes has to exclude such marks from hover rather than rank them wrongly.

Distances are: to the disc for round marks (zero inside), to the rectangle for rects (zero inside), perpendicular to the segment for segments, to the nearest edge for open polylines, and **zero anywhere inside** a closed polygon or path — so clicking the middle of a choropleth region hits that region.

Marks vellum cannot describe more precisely than a box — labels, rounded rects — fall back to their box, which for a label is a fair description of the target anyway.

For a client-side host

A browser cannot call back into R on every mouse move. Use `element_geometry()` instead: it returns the same true geometry once, and the client computes distances locally. `vl_nearest()` is for hosts that *can* ask — a Shiny app, a test, a script.

See Also

`element_geometry()` for the geometry itself, `scene_model()` for bounding boxes, `hit_test()` for "which grob is under this pixel".

Examples

```
s <- vl_scene(4, 3, dpi = 96, bg = "white") |>
  draw(segments_grob(0.1, 0.1, 0.9, 0.9, key = "diagonal")) |>
  draw(points_grob(0.8, 0.2, key = "corner"))

# Near the corner the diagonal's *bounding box* also contains this point,
# but its geometry is far away.
vl_nearest(s, 0.8, 0.2)
```

vl_path_op

Boolean operations on paths

Description

Combine two shapes as **geometry**: union, intersection, difference or exclusive-or over their closed rings.

Usage

```
vl_path_op(
  a,
  b,
  op = c("union", "intersect", "difference", "xor"),
  rule = c("nonzero", "evenodd"),
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  role = NULL
)
```

Arguments

a, b	The operands: a <code>path_grob()</code> , a <code>polygon_grob()</code> , or a list with x, y and optionally <code>nper</code> (points per ring).
op	One of "union", "intersect", "difference" (a minus b), or "xor".
rule	Fill rule for interpreting the inputs: "nonzero" (default) or "evenodd".
gp, name, vp, role	Passed to the returned <code>path_grob()</code> .

Value

A `path_grob()` with `rule = "winding"`. An empty result (a disjoint intersection, say) gives a grob with no points, which draws nothing.

Why geometry and not a mask

vellum can already clip one shape by another at render time, and for simply *showing* an intersection that is often enough. A boolean result is different in kind: it is an ordinary path, so it can be measured, hit-tested, simplified, filled with a gradient, stroked, exported as `<path>` data, and fed into another boolean. A mask can do none of those, rasterises, and degrades on some PDF paths.

Rings, holes and the fill rule

`rule` says how to interpret the **inputs** — whether a ring inside another ring is a hole ("`evenodd`") or a separate island ("`nonzero`"). It must match the rule the operands were drawn with, or the answer will be correct for a shape you did not mean.

The **result** always uses the non-zero rule: holes come back wound opposite to their outer ring, which is how the returned `path_grob()` is set up.

Operands must be in a single coordinate space — one unit, no offsets — since a boolean has to be computed somewhere, and mixed units have no common space.

Examples

```
sq <- function(x0, y0, s = 0.4) {
  list(x = c(x0, x0 + s, x0 + s, x0), y = c(y0, y0, y0 + s, y0 + s))
}
a <- sq(0.2, 0.3)
b <- sq(0.45, 0.4)
vl_scene(3, 2, dpi = 96, bg = "white") |>
  draw(vl_path_op(a, b, "union", gp = vl_gpar(fill = "#DCE7F5", col = "steelblue")))
```

vl_pattern

Tiling-pattern fills

Description

Create a pattern that fills a shape by tiling a grob. The grob is drawn once into a tile occupying the unit square (0..1 npc), then repeated across a cell of size **width** x **height** (in **units**) anchored at (x, y). Like gradients, the cell geometry is resolved against the viewport at draw time.

Usage

```
vl_pattern(
  grob,
  width = 0.1,
  height = 0.1,
  x = 0.5,
  y = 0.5,
  units = "npc",
  extend = "repeat"
)
```

Arguments

<code>grob</code>	A grob, or a list of grobs, drawn into the tile (their 0..1 npc coordinates map to the tile, painted in order).
<code>width, height</code>	Size of one tile cell (default 0.1 npc).
<code>x, y</code>	Cell centre (default centred).
<code>units</code>	Coordinate system for the geometry; see linear_gradient() .
<code>extend</code>	Tiling mode: "repeat" (default), "reflect" , or "pad" . (SVG renders all modes as repeat .)

Details

The tile is rendered to a raster image (sized from `width/height` at the scene's resolution) and embedded on every backend: PNG raster, SVG `<image>` in a `<pattern>`, and a PDF tiling pattern with the tile as an embedded image XObject. Only a degenerate tile or cell size fails, leaving the shape unfilled with a degrade warning.

Value

A `vellum_pattern` object, suitable for `vl_gpar(fill = ...)`.

Examples

```
dots <- circle_grob(r = 0.25, gp = vl_gpar(fill = "white", col = NA))
vl_pattern(dots, width = 0.08, height = 0.08)
```

`vl_place`

Move labels so they stop overlapping

Description

`vl_place()` solves label collisions over a scene's **resolved** geometry and reports the displacement each label needs. `vl_repel()` applies that solution and returns a new scene.

Usage

```
vl_place(
  scene,
  labels = NULL,
  avoid = NULL,
  padding = 1,
  max_shift = 10,
  max_iter = 200,
  pull = 0.1
)

vl_repel(
  scene,
  labels = NULL,
  avoid = NULL,
  padding = 1,
  max_shift = 10,
  max_iter = 200,
  pull = 0.1
)
```

Arguments

scene	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method).
labels	Names of the grobs to move. <code>NULL</code> (default) takes every text node in the scene.
avoid	Names of the grobs to treat as obstacles. <code>NULL</code> (default) takes every node that is not a label.
padding	Clear space to keep around each label, in millimetres.
max_shift	The furthest a label may travel from its anchor, in millimetres. Labels needing more are left at their best position and reported as unresolved.
max_iter	Iteration cap for the relaxation.
pull	Strength of the spring back to the anchor, in <code>[0, 1]</code> . Higher keeps labels closer to what they label; lower resolves crowding more readily.

Value

`vl_place()`: a data frame with one row per label — **name**, **index** (which element of that grob), the original device box `x0/y0/x1/y1`, the shift `dx/dy` in millimetres, and **resolved** (did it end up clear). `vl_repel()`: a new scene.

Why this is an engine service

Repulsion is a geometry problem over boxes: where things ended up, how big they are, and what they must not touch. It carries no data semantics, so every layer above vellum would otherwise reimplement it — and reimplement it against numbers it can only get by rendering first.

Because the solve happens in **device pixels** and the answer is applied as an absolute millimetre offset on top of each label's existing coordinate, it does not care what produced the position. A label anchored in **native** units inside a polar viewport, a facet panel, or a warped coordinate system moves by the same mechanism as one in **npc** on the page, and panels are all solved together rather than one at a time.

What it does not do

It moves labels; it does not choose which to drop, shrink or abbreviate — those are editorial decisions belonging to the layer that knows what the labels mean. Labels that cannot be separated within **max_shift** are reported with **resolved = FALSE** rather than silently piled up or quietly deleted.

See Also

[vl_empty_region\(\)](#) for placing one thing in the gap rather than moving many things apart.

Examples

```
set.seed(1)
n <- 12
s <- vl_scene(5, 3, dpi = 96, bg = "white") |>
  draw(points_grob(runif(n), runif(n), gp = vl_gpar(fill = "steelblue", col = NA),
    name = "pts")) |>
  draw(text_grob(paste0("item ", seq_len(n)), x = runif(n), y = runif(n),
    gp = vl_gpar(fontsize = 9), name = "lab"))
head(vl_place(s))
```

`vl_render_animation` *Render a keyframe animation*

Description

Interpolate between a set of compiled keyframe scenes and encode the in-between frames to an animated image, in one parallel, streaming pass in the Rust backend. This is the low-level engine a grammar layer (e.g. `vellumplot`'s `animate()`) drives: the caller supplies the K keyframe scenes and a per-frame *schedule* — for each output frame, which adjacent keyframe pair to interpolate and the eased fraction between them — and this renders and encodes every frame.

Usage

```
vl_render_animation(
  keyframes,
  seg,
  frac,
  path,
  format = c("gif", "apng", "svg", "frames"),
  fps = 25,
```

```

    gif_speed = 1,
    gif_dither = TRUE,
    frac_col = NULL,
    frac_size = NULL,
    frac_alpha = NULL
)

```

Arguments

keyframes	A list of at least two scenes (each a <code>vl_scene()</code> or anything with an <code>as_vellum_scene()</code> method), all the same pixel size.
seg	Integer vector, one entry per output frame: the 1-based index of the frame's left keyframe (it is interpolated with keyframe <code>seg + 1</code>).
frac	Numeric vector, one per output frame: the eased interpolation fraction in <code>[0, 1]</code> (0 = the left keyframe, 1 = the right one). Same length as <code>seg</code> . This is the schedule for positional geometry, and for every discrete attribute that snaps at the halfway point.
path	Output path: an image file for <code>"gif"/"apng"/"svg"</code> , or a directory (created if needed) for <code>"frames"</code> .
format	<code>"gif"</code> (looping animated GIF), <code>"apng"</code> (animated PNG), <code>"svg"</code> (a single animated SVG), or <code>"frames"</code> (one <code>frameNNNNN.png</code> per frame into <code>path</code>).
fps	Frames per second (sets each frame's on-screen duration).
gif_speed	GIF only: the NeuQuant palette sample factor, 1 (best quality, slowest) to 30 (fastest). A plot's antialiased edges want the best palette, so the default is 1. Ignored for <code>"apng"/"frames"</code> .
gif_dither	GIF only: apply Floyd–Steinberg dithering (default <code>TRUE</code>), which greatly reduces the banding a 256-colour palette leaves on gradients and antialiased edges. A frame that already fits in 256 colours is kept exact.
frac_col, frac_size, frac_alpha	Optional per-aesthetic schedules, each a numeric vector the same length as <code>frac</code> , for the colour, size and opacity classes respectively. <code>NULL</code> (the default) uses <code>frac</code> , so one easing curve shapes the whole scene. Supplying a differently-eased vector lets each aesthetic travel on its own curve — position arriving on cubic-in-out while colour crossfades linear , say. See <i>Per-aesthetic easing</i> below.

Details

The frames are tweened at the scene level: matching primitives interpolate their geometry, their colours (perceptually, in Oklab), and their bounded graphical parameters; discrete attributes snap. Nothing retrains between frames — the keyframes are fixed at author time — so this is non-reactive keyframe animation, not a live/reactive runtime.

GIF is limited to 256 colours per frame, so on a plot (smooth panels, antialiased marks) it is inherently lossy — `gif_speed/gif_dither` make it as clean as that palette allows. For a lossless result use `format = "apng"`.

Value

path, invisibly.

Choosing a format

"svg" emits every frame as vector markup, shown in turn by a CSS step animation. It is resolution-independent, which no raster format is — the same file is crisp in a slide, on a retina screen and in print.

Its size depends on scene complexity in a way the raster formats' does not, because *every frame is emitted in full*. Measured on a 30-frame scatter animation, gzipped (which is how a browser will fetch it):

marks	animated SVG (gzipped)	GIF
20	20 KB	61 KB
200	80 KB	296 KB
2000	720 KB	124 KB

So it wins clearly on line art — an explanatory animation of a few moving marks, which is the common case — and loses on a dense scatter, where a raster format is the right answer. Serve it gzipped (`.svgz`, or any web server with compression on); uncompressed it is several times larger again.

It also honours `prefers-reduced-motion`: a reader who has asked their system not to animate gets the first frame, held.

Per-aesthetic easing

`frac` and its three companions carry one eased fraction per **property class**, so a single frame can interpolate different properties at different points along their transitions. Every drawn property belongs to exactly one class:

<code>schedule</code>	<code>drives</code>
<code>frac</code>	x/y and all coordinate geometry, widths and heights, angles, path vertices, text rotation, dash phase
<code>frac_col</code>	<code>fill</code> , <code>col</code> , the stroke paint, and per-element colour vectors (hexagon and sector fills)
<code>frac_size</code>	<code>lwd</code> , marker size, circle and corner radius, hexagon extent, <code>linemitre</code>
<code>frac_alpha</code>	<code>alpha</code> , including the enter/exit fade — a keyed element appearing or leaving fades on this curve

Two consequences worth knowing. Easing `alpha` retimes entrances and exits, not just explicit opacity changes. And because a discrete attribute flips when its fraction crosses 0.5, and an eased curve reaches 0.5 at a different *frame* than a linear one, `frac` decides which frame those snaps land on.

Passing all four identical (the default) reproduces single-curve easing exactly — the output is byte-identical to omitting them.

See Also

`render()`, `as_vellum_scene()`

Examples

```
## Not run:
grow <- lapply(c(0.1, 0.3, 0.2), function(r) {
  vl_scene(3, 2) |> draw(circle_grob(r = r, gp = vl_gpar(fill = "tomato")))
})
# 30 frames across the 3 keyframes, held on the last one.
seg <- rep(1:2, each = 15)
frac <- rep(seq(0, 1, length.out = 15), 2)
vl_render_animation(grow, seg, frac, tempfile(fileext = ".gif"))

## End(Not run)
```

vl_scene

Build and render a scene

Description

`vl_scene()` creates an empty scene. `push()` adds a viewport (and descends into it), `draw()` adds a grob, `pop()` ascends. The builder is a linear pipe; a *rendered or edited* scene is an immutable value (`edit_node()` copies on modify). `render()` compiles the scene and writes the output.

Usage

```
vl_scene(
  width = 6,
  height = 4,
  dpi = 96,
  bg = "white",
  gp = vl_gpar(),
  xscale = c(0, 1),
  yscale = c(0, 1),
  clip = FALSE,
  title = NULL,
  desc = NULL
)

push(scene, vp)

draw(scene, grob)

pop(scene, n = 1)

render(
  scene,
  path,
  text = c("native", "outline"),
```

```

    debug = FALSE,
    scale = 1,
    cvd = "none"
)

```

Arguments

<code>width, height</code>	Page size (vl_unit() or numeric inches).
<code>dpi</code>	Resolution in dots per inch.
<code>bg</code>	Background colour (or NA for transparent).
<code>gp</code>	Page-level graphical parameters (vl_gpar()) carried by the root viewport; inherited by everything drawn (e.g. a default <code>col/fontsize</code>).
<code>xscale, yscale</code>	Native coordinate range of the root viewport, so "native" units work at the page level without an explicit push() .
<code>clip</code>	Clip drawing to the page rectangle?
<code>title, desc</code>	Accessibility: an accessible name (a short title) and a longer description (alt text) for the scene. When either is set, the SVG backend emits <code>role="img" + <title>/<desc></code> (referenced by <code>aria-labelledby</code>) and the PDF backend tags the page as a Figure with the description as Alt text. NULL (default) emits no accessibility markup, so output is unchanged. See describe() to set them on an existing scene.
<code>scene</code>	A vl_scene() .
<code>vp</code>	A vl_viewport() .
<code>grob</code>	A grob (see grob).
<code>n</code>	Number of viewport levels to ascend.
<code>path</code>	Output file path; the format is taken from the extension (<code>.png</code> , <code>.svg</code> , or <code>.pdf</code>).
<code>text</code>	For SVG output, how text is written: "native" (default) emits selectable <code><text></code> referencing system fonts, "outline" emits glyph outlines (pixel-faithful, identical to the raster/PDF backends, but not selectable). Ignored for PNG/PDF.
<code>debug</code>	If TRUE, overlay a layout-debug skeleton on the output: each viewport region (outlined and labelled by name), its layout track boundaries, and its clip region. Built from the resolved scene with why_size() ; useful for understanding why elements land where they do. Default FALSE.
<code>scale</code>	Resolution multiplier (default 1). <code>scale = 2</code> renders at twice the device pixels while keeping the same physical size — the retina / <code>ggsave(scolding=)</code> idiom. It multiplies <code>dpi</code> , so absolute units (<code>mm</code> , <code>pt</code> , <code>in</code>) cover proportionally more pixels and nothing about the layout changes; text does not get relatively bigger or smaller. Only raster output gains anything: a PDF's page size in points and an SVG's physical size are unchanged by construction.
<code>cvd</code>	Simulate a colour-vision deficiency: "none" (default), "protanopia" (red-blind), "deutanopia" (green-blind, the common one that breaks red/green

encodings), "tritanopia" (blue-blind), or "achromatopsia" (total, and a fair proxy for greyscale printing). The simulation is a post-pass over the finished raster using the Machado et al. (2009) matrices applied in linear light, so it turns an accessibility check into a one-line change instead of an export-and-upload round trip. **Raster output only** — a vector format has no pixels to transform, and `render()` warns if you ask for one.

Value

`vl_scene()`, `push()`, `draw()`, `pop()`: a `vellum_scene`.
`render()`: path, invisibly.

Examples

```
s <- vl_scene(width = 4, height = 3) |>
  push(vl_viewport(xscale = c(0, 10), yscale = c(0, 10))) |>
  draw(rect_grob(gp = vl_gpar(fill = "grey95", col = "grey50"))) |>
  draw(lines_grob(x = vl_unit(0:10, "native"), y = vl_unit(0:10, "native"),
    gp = vl_gpar(col = "steelblue", lwd = 2)))
```

vl_shadow

Drop shadow for a viewport group

Description

Describes a shadow cast by everything drawn inside a `vl_viewport()`: the group's own silhouette, tinted, blurred and offset, painted underneath it. Because it is a *group* effect, overlapping shapes inside the viewport cast one shadow together rather than each casting its own onto the others.

Usage

```
vl_shadow(dx = 2, dy = 2, blur = 3, col = "#00000059")
```

Arguments

`dx`, `dy` Offset in points; positive `dy` moves the shadow down.
`blur` Blur radius in points. 0 gives a hard offset silhouette.
`col` Shadow colour; usually a translucent black.

Value

A `vellum_shadow` object, for `vl_viewport(shadow = )`.

See Also

[vl_viewport\(\)](#)

Examples

```
vl_scene(3, 2) |>
  push(vl_viewport(width = 0.6, height = 0.6, shadow = vl_shadow())) |>
  draw(circle_grob(gp = vl_gpar(fill = "steelblue", col = NA))) |>
  pop()
```

vl_strwidth

Measure text

Description

`vl_strwidth()` / `vl_strheight()` return the rendered width/height of each string, using the same shaping (**textshaping**/HarfBuzz + **systemfonts**) the renderer uses, so measurements match drawn text. Device-independent (does not need an open scene). Vectorised over `label`. (Named `vl_*` to avoid masking `grDevices::strwidth()`.)

Usage

```
vl_strwidth(
  label,
  family = "",
  fontface = "plain",
  fontsize = 12,
  cex = 1,
  unit = "in",
  features = NULL
)

vl_strheight(
  label,
  family = "",
  fontface = "plain",
  fontsize = 12,
  cex = 1,
  unit = "in",
  features = NULL
)
```

Arguments

label	Character vector of strings to measure, or a rich label from <code>md()</code> (or a list of them). Rich labels are measured with the same run composition the renderer draws, so super/subscripts and bold runs are accounted for; family / fontface / fontsize then supply the <i>base</i> style the label's runs are relative to.
family	Font family (e.g. <code>"sans"</code> , <code>"serif"</code> , <code>"mono"</code> , or a specific family name). <code>""</code> uses the system default.

fontface	One of "plain", "bold", "italic", "bold.italic".
fontsize	Font size in points.
cex	Multiplier applied to fontsize .
unit	Output unit: one of "in", "pt", "mm", "cm".
features	OpenType features as a named vector of four-character tags (e.g. <code>c(tnum = 1)</code>), matching <code>vl_gpar()</code> 's features . Measurement must use the same features as drawing, or reserved space will not match.

Value

A numeric vector (one per **label**) of widths/heights in **unit**.

Examples

```
vl_strwidth(c("short", "a longer label"), fontsize = 14)
```

<code>vl_svg_path</code>	<i>SVG path data as scene geometry</i>
--------------------------	--

Description

`vl_svg_path()` parses the **d** attribute of an SVG `<path>` into rings; `svg_grob()` wraps that up as a drawable `path_grob()`.

Usage

```
vl_svg_path(d)

svg_grob(
  d,
  x = 0.5,
  y = 0.5,
  size = vl_unit(10, "mm"),
  flip_y = TRUE,
  viewBox = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)
```

Arguments

d	A character string of SVG path data, or (for <code>svg_grob()</code>) a whole <code><svg></code> element to read <code><path></code> geometry and the <code>viewBox</code> from.
x, y	Centre of the drawn icon.
size	Size of the reference box's longer side, as a <code>vl_unit()</code> — the <code>viewBox</code> when one is given, otherwise the path's own bounding box.
flip_y	Flip the y axis to convert from SVG's convention. Leave <code>TRUE</code> unless your <code>d</code> is already in a y-up space.
viewbox	The icon's SVG <code>viewBox</code> as <code>c(xmin, ymin, width, height)</code> or the attribute string <code>"xmin ymin width height"</code> , so <code>size</code> scales the box rather than the glyph's ink. <code>NULL</code> (default) uses a whole- <code><svg></code> document's own <code>viewBox</code> if present, else the path's ink bounds.
gp, name, vp, id, role	Passed to the returned <code>path_grob()</code> .

Details

This is what makes vector icons usable as marks. Icon sets — Font Awesome, Bootstrap Icons, Lucide, Material — ship one `<path d="...">` per glyph, so `d` is the unit of exchange, and the result here is real geometry: crisp at any size, fillable with a gradient, strokable, and exported as `<path>` data rather than an embedded bitmap.

Value

`vl_svg_path()`: a data frame of `x`, `y`, `id`, `closed`. `svg_grob()`: a `path_grob()`.

Coordinate system

SVG's y axis points **down** and vellum's points up, so `svg_grob()` flips it by default (`flip_y = TRUE`) — otherwise every icon arrives upside-down. The geometry is then scaled so the longer side of its reference box maps to `size`, preserving aspect, and centred on `x/y`. `vl_svg_path()` returns the raw parsed coordinates without any of that, for callers doing their own placement.

Sizing across an icon set — the viewBox

Icon sets draw every glyph inside one shared `viewBox` (commonly `"0 0 24 24"`) and pad each glyph within it, so a glyph's own ink fills only part of the box. Sizing to the *ink* would blow each glyph up to `size` individually — a sparse glyph would render far larger than a dense one from the same set, and a lone icon larger than its nominal box. Pass the set's `viewbox` (the four numbers `c(xmin, ymin, width, height)`, or the raw attribute string `"0 0 24 24"`) and `size` maps the *box* instead, so every glyph keeps its intended relative and absolute size. If you hand `svg_grob()` a whole `<svg>...</svg>` element (see below) its `viewBox` is used automatically. With no `viewBox` at all, sizing falls back to the path's own ink bounds.

What is supported

The whole `d` grammar: M/L/H/V/C/S/Q/T/A/Z in absolute and relative forms, implicit repeated commands, the smooth-curve reflection rules, and elliptical arcs. Curves are flattened to polylines.

`svg_grob()` also accepts a whole `<svg>` element as `d`: it reads the `<path>` geometry and the document `viewBox` (needs the `xml2` package). Only `<path>` elements are read — other drawable shapes (`<circle>`, `<rect>`, `<line>`, ...) are not path data and are reported with a warning, not silently dropped. The rest of SVG is still out of scope: no stylesheets, gradients, `<use>`, clip paths, or element transforms — this reads geometry, not documents.

Malformed data yields whatever parsed before the problem rather than an error: a truncated icon is easier to diagnose than a stack trace.

Examples

```
# A five-pointed star, as an icon set would ship it.
star <- "M12 2 L15 9 L22 9.3 L16.5 13.8 L18.5 21 L12 17 L5.5 21 L7.5 13.8 L2 9.3 L9 9 Z"
vl_scene(3, 2, dpi = 96, bg = "white") |>
  draw(svg_grob(star, x = 0.5, y = 0.5, size = vl_unit(15, "mm"),
    gp = vl_gpar(fill = "#F1C40F", col = "grey30"))))

# As per-point markers, which is what raster icons cannot do crisply.
set.seed(1)
s <- vl_scene(4, 2, dpi = 96, bg = "white")
for (i in 1:6) {
  s <- draw(s, svg_grob(star, x = i / 7, y = runif(1) * 0.6 + 0.2,
    size = vl_unit(6, "mm"),
    gp = vl_gpar(fill = "#2C6FA6", col = NA)))
}
s
```

vl_unit

Units of measurement

Description

A `unit` is a vectorised (`value`, `unit`) pair used for coordinates and sizes. Each element carries its own unit, so a single `unit` vector can mix coordinate systems and a primitive can use different units on the x and y axes (e.g. x in `"native"`, y in `"npc"`).

Usage

```
vl_unit(values, units = "npc", data = NULL)

is_unit(x)
```

Arguments

<code>values</code>	Numeric vector of magnitudes.
<code>units</code>	Character vector of unit names, recycled against <code>values</code> .
<code>data</code>	Optional list supplying context for derived units: <code>label</code> , <code>fontfamily</code> , <code>fontface</code> , <code>fontsize</code> , <code>lineheight</code> .
<code>x</code>	An object.

Details

Supported units:

- `"npc"` — normalised parent coordinates (0 = bottom/left, 1 = top/right)
- `"native"` — the enclosing viewport's `xscale/yscale`
- `"mm"`, `"cm"`, `"in"`, `"pt"` — absolute lengths
- `"char"`, `"line"` — font-relative (need `fontsize/lineheight` via `data`)
- `"strwidth"`, `"strheight"` — size of a string (need `label` via `data`)

Font- and string-relative units are resolved to absolute millimetres at construction (text metrics are available device-independently), so a stored `unit` only ever holds one of the core backend units.

Arithmetic: `+` and `-` combine two units of the *same* code, or two *absolute* units (`"mm"/"cm"/"in"/"pt"`), which resolve to `"mm"` immediately (e.g. `vl_unit(10, "mm") + vl_unit(1, "in")` is 35.4mm). A position unit (`"npc"/"native"`) plus an absolute unit forms a **compound** unit — a `data/panel` anchor plus an exact absolute offset — e.g. `vl_unit(1, "native") + vl_unit(2, "mm")` is `1native+2mm`: it resolves to the native position shifted right by exactly 2 mm at render, at any scale or aspect. Mixing two *different* position bases (e.g. `"npc"` and `"native"`) still errors, as it cannot be reduced to one unit. `unit * scalar` scales the base value and the offset together.

Value

A `unit` vector.

Examples

```
vl_unit(1:3, "native")
vl_unit(c(0.5, 1), c("npc", "in"))
```

vl_viewport
Viewports and layouts

Description

A viewport is a rectangular region that establishes its own coordinate systems (its `xscale/yscale` for "native" units), optionally rotated, clipped, and carrying inheritable graphical parameters. Push one onto a scene with `push()`. A viewport may define a row/column `grid_layout()`; child viewports are then placed into cells via `row/col`.

Usage

```
vl_viewport(
  x = 0.5,
  y = 0.5,
  width = 1,
  height = 1,
  xscale = c(0, 1),
  yscale = c(0, 1),
  angle = 0,
  clip = FALSE,
  gp = vl_gpar(),
  layout = NULL,
  row = NULL,
  col = NULL,
  rowspan = 1,
  colspan = 1,
  mask = NULL,
  alpha = NULL,
  blend = NULL,
  blur = NULL,
  shadow = NULL,
  name = NULL,
  meta = NULL,
  pannable = FALSE,
  cache = FALSE
)

grid_layout(
  widths = vl_unit(1, "null"),
  heights = vl_unit(1, "null"),
  respect = FALSE
)
```

Arguments

`x, y` Centre of the viewport (`vl_unit()` or numeric, in the parent).

<code>width, height</code>	Size (<code>vl_unit()</code> or numeric, in the parent).
<code>xscale, yscale</code>	Length-2 native coordinate ranges.
<code>angle</code>	Rotation in degrees, counter-clockwise about the centre.
<code>clip</code>	Clip drawing to this viewport: <code>TRUE/FALSE</code> for the viewport rectangle, or a <code>polygon_grob()/path_grob()</code> (in this viewport's coordinates) to clip to an arbitrary path.
<code>gp</code>	Inheritable graphical parameters, from <code>vl_gpar()</code> .
<code>layout</code>	An optional <code>grid_layout()</code> .
<code>row, col</code>	Cell (1-based) of the parent's layout to place into.
<code>rowspan, colspan</code>	Number of cells to span.
<code>mask</code>	An optional mask: a grob (or list of grobs), or an <code>as_mask()</code> result. The viewport's contents are rendered as an isolated layer and the mask modulates their visibility.
<code>alpha</code>	Optional group opacity in <code>[0, 1]</code> . The viewport's contents are composited as a single isolated layer at this opacity, so overlapping elements do not accumulate (unlike per-element <code>vl_gpar(alpha=)</code>). <code>NULL</code> (default) means fully opaque.
<code>blend</code>	Optional blend mode for compositing the viewport's contents (as an isolated layer) onto the backdrop below it. One of <code>"normal"</code> (default), <code>"multiply"</code> , <code>"screen"</code> , <code>"overlay"</code> , <code>"darken"</code> , <code>"lighten"</code> , <code>"color-dodge"</code> , <code>"color-burn"</code> , <code>"hard-light"</code> , <code>"soft-light"</code> , <code>"difference"</code> , <code>"exclusion"</code> , <code>"hue"</code> , <code>"saturation"</code> , <code>"color"</code> , or <code>"luminosity"</code> (the CSS <code>mix-blend-mode</code> set). <code>NULL/"normal"</code> is ordinary over-compositing.
<code>blur</code>	Optional Gaussian blur radius in points, applied to the viewport's contents as one isolated layer. Because it is a <i>group</i> effect, overlapping shapes inside the viewport blur together rather than each blurring over the others. <code>NULL/0</code> (default) means no blur. Raster convolves it; SVG emits a native <code>feGaussianBlur</code> and stays vector; PDF has no filter model and warns.
<code>shadow</code>	Optional drop shadow from <code>vl_shadow()</code> , cast by the viewport's contents as one silhouette. A shadow with no offset is a glow. Same per-backend behaviour as <code>blur</code> .
<code>name</code>	Optional name (for <code>edit_node()</code>).
<code>meta</code>	Optional free-form metadata for this viewport (any R object, default <code>NULL</code>). Like a grob's <code>meta</code> , it never crosses to the rendering backend — it rides on the R scene and surfaces, for a <i>named</i> viewport, as the <code>meta</code> column of <code>scene_model()</code> 's <code>panels</code> table. A host (e.g. <code>vellumwidget</code>) reads it; <code>vellum</code> neither inspects nor validates it. This is the panel-level counterpart of the per-element grob <code>meta</code> channel, intended for panel-scoped conventions such as axis/scale descriptors.
<code>pannable</code>	Emit this named panel as a clip-stable pannable group (default <code>FALSE</code>): an outer <code><g data-vellum-panel></code> carrying the panel's clip (untransformed, so the clip stays fixed) wrapping an inner <code><g data-vellum-pan></code> that

holds the content. A host (e.g. `vellumwidget`) can set a `transform` on the inner group to pan/zoom the marks while the clip and the surrounding axes stay put. SVG only; requires a named viewport. No effect on the rendered (static) output — the extra inner group is inert until a host transforms it.

cache Repaint boundary (TRUE/FALSE, default FALSE). Flag this viewport's subtree as a cached sub-raster: on render it is rasterised once to its own layer and, on later renders where the subtree is **unchanged**, the cached pixels are composited instead of re-drawing the subtree. This makes partial redraw cheap — highlight/hover (`edit_node()` one element and re-render) or animation (one subtree changes, others static) re-rasterise only what changed. Raster/`display()` only; SVG/PDF ignore it and render the subtree as vector (no fidelity loss). Ignored when the viewport also sets a non-normal `blend` (a blend needs the live backdrop). See `vl_clear_render_cache()`.

widths, heights Track sizes as a `vl_unit()` vector. Use "null" units for flexible tracks that share leftover space in proportion to their value.

respect Logical; if TRUE, lock the layout's aspect grid-style: one unit of "null" width is forced to the same physical (device) size as one unit of "null" height. The axis whose null unit would be larger shrinks to match and the whole grid is centered in its parent (so absolute gutter tracks stay attached to the flexible cells). Encode a desired cell aspect in the null track weights — a cell of null width-weight `w` by height-weight `h` then renders with device aspect `w:h`. Default FALSE (tracks just fill the parent). This is how a fixed-aspect panel (e.g. `coord_fixed()`, `maps`) is built on top of vellum.

Value

A viewport object.

`grid_layout()`: a layout object.

Examples

```
vl_viewport(xscale = c(0, 10), yscale = c(0, 100))
```

why_size

Explain why a node has its resolved size

Description

Reports the resolved width and height of a named viewport (or grob) and what determined them — the layout track it was placed in, or the units of its own width/height. The layout companion to the visual `render(scene, debug = TRUE)` overlay; together they make coordinate debugging first-class rather than an exercise in archaeology.

Usage

```
why_size(scene, name)
```

Arguments

<code>scene</code>	A <code>vl_scene()</code> (or anything with an <code>as_vellum_scene()</code> method).
<code>name</code>	A node name (set via the <code>name</code> argument of a viewport/grob).

Value

A `vellum_why_size` record (a list with `name`, `width_mm`, `height_mm`, and `determined_by`), printed legibly.

Examples

```
s <- vl_scene(4, 3) |>
  push(vl_viewport(name = "panel", width = vl_unit(2, "in"), height = vl_unit(1, "in")))
why_size(s, "panel")
```

Index

as_mask, 3
as_mask(), 84
as_scene_spec, 4
as_scene_spec(), 34–36, 42
as_vellum, 5
as_vellum_scene, 6
as_vellum_scene(), 4, 12–14, 16, 31,
33–35, 38–40, 42, 54, 56, 61, 63,
64, 67, 71, 73, 74, 86

bezier_grob (grob), 19
bezier_grob(), 50

circle_grob (grob), 19
circle_grob(), 42
contour_grob (vl_contour), 52

datashade, 7
datashade(), 9, 11, 14, 40, 43, 44, 52
datashade_lines, 9
datashade_lines(), 8, 14, 43, 44
datashade_segments (datashade_lines),
9
datashade_segments(), 8, 43
describe, 12
describe(), 76
display, 12
display(), 85
draw (vl_scene), 75
draw(), 8, 11, 19, 75
dynspread, 13
dynspread(), 8, 11, 43, 44

edit_node (node_names), 30
edit_node(), 24, 36, 49, 75, 84, 85
element_geometry, 14
element_geometry(), 68

font_check (font_pin), 16
font_pin, 16
font_pin(), 34, 62

from_scene_spec (as_scene_spec), 4

get_node (node_names), 30
gradients, 17
graphics::contour(), 53
graphics::image(), 53
graphics::persp(), 53
graphics::plot(), 39
grDevices::as.raster(), 27, 39
grid::rasterGrob(), 39
grid_layout (vl_viewport), 83
grid_layout(), 28, 83, 84
grob, 8, 11, 14, 19, 44, 46, 76
grobheight (grobwidth), 27
grobheight(), 29, 48, 57
grobwidth, 27
grobwidth(), 29, 48, 57

hexagon_grob (grob), 19
hexagon_grob(), 24
hit_test, 28
hit_test(), 67, 68

is_unit (vl_unit), 81

linear_gradient (gradients), 17
linear_gradient(), 47, 57, 59, 70
lines_grob (grob), 19
lines_grob(), 25, 42, 45, 50, 53
loop_grob (grob), 19
loop_grob(), 24, 27

md, 29
md(), 78

node_names, 30
node_names(), 36

path_grob (grob), 19
path_grob(), 45, 46, 68, 69, 79, 80, 84
pdf_pages, 30

pdf_pages(), 33
 points_grob (grob), 19
 polygon_grob (grob), 19
 polygon_grob(), 42, 45, 60, 68, 84
 pop (vl_scene), 75
 pop(), 75
 profile_render, 31
 profile_render(), 40
 push (vl_scene), 75
 push(), 75, 76, 83

 radial_gradient (gradients), 17
 radial_gradient(), 47, 57
 raster_grob (grob), 19
 raster_grob(), 7–9, 11
 rect_grob (grob), 19
 rect_grob(), 42
 render (vl_scene), 75
 render(), 5, 6, 19, 31–33, 38, 39, 41, 59, 74, 75
 render_all, 32
 render_grid (as_vellum), 5
 roundrect_grob (grob), 19

 scene_diff, 33
 scene_diff(), 5, 35, 42
 scene_fonts, 34
 scene_hash, 35
 scene_hash(), 5, 34, 42, 62
 scene_inset, 35
 scene_model, 37
 scene_model(), 15, 24, 25, 41, 54, 55, 68
 scene_pdf (scene_png), 38
 scene_png, 38
 scene_png(), 41
 scene_raster, 39
 scene_raster(), 38, 39
 scene_read (scene_write), 41
 scene_stats, 40
 scene_stats(), 32, 62
 scene_svg, 41
 scene_svg(), 37–39
 scene_write, 41
 scene_write(), 5
 sector_grob (grob), 19
 segments_grob (grob), 19
 segments_grob(), 25, 46, 50
 sketch, 42
 sketch(), 24

 spline_grob (grob), 19
 spread, 43
 spread(), 8, 11, 13, 14
 stroke_to_path, 44
 stroke_to_path(), 25, 26
 style, 46
 svg_grob (vl_svg_path), 79

 text_grob (grob), 19
 text_grob(), 24, 29
 text_path_grob, 49

 vl_arrow, 50
 vl_arrow(), 25, 27
 vl_buffer, 51
 vl_clear_render_cache, 52
 vl_clear_render_cache(), 85
 vl_contour, 52
 vl_convert, 54
 vl_empty_region, 55
 vl_empty_region(), 72
 vl_gpar, 56
 vl_gpar(), 17, 24, 45, 46, 49, 76, 79, 84
 vl_hatch, 58
 vl_hull, 59
 vl_hull(), 51
 vl_lint, 60
 vl_lint(), 32, 40, 59, 63–66
 vl_lint_assert, 62
 vl_lint_assert(), 64
 vl_lint_finding, 63
 vl_lint_finding(), 65
 vl_lint_overlay, 64
 vl_lint_overlay(), 63
 vl_lint_rule, 65
 vl_lint_rule(), 62, 63
 vl_lint_rules (vl_lint_rule), 65
 vl_lint_rules(), 66
 vl_nearest, 67
 vl_nearest(), 15
 vl_path_op, 68
 vl_pattern, 69
 vl_pattern(), 58, 59
 vl_place, 70
 vl_render_animation, 72
 vl_repel (vl_place), 70
 vl_scene, 75

`vl_scene()`, 4–6, 12–14, 16, 29–31, 33–35,
37–42, 54, 56, 61, 63, 64, 67, 71,
73, 76, 86

`vl_shadow`, 77

`vl_shadow()`, 84

`vl_strheight` (*vl_strwidth*), 78

`vl_strwidth`, 78

`vl_svg_path`, 79

`vl_unit`, 81

`vl_unit()`, 19, 24–26, 28, 51, 54, 55, 76,
80, 83–85

`vl_viewport`, 83

`vl_viewport()`, 8, 11, 24, 28, 36, 49, 76,
77

`why_size`, 85

`why_size()`, 54, 55, 62, 76