Package: vellum (via r-universe)

July 12, 2026

Title A Low-Level Graphics Framework with a Rust Backend

Version 0.2.0.9000

Description A grid-like low-level graphics system for R whose 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 testthat (>= 3.0.0), withr, ggplot2, knitr, png, base64enc, rmarkdown

VignetteBuilder knitr

Config/testthat/edition 3

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

Collate 'gpar.R' 'viewport.R' 'grob.R' 'api.R' 'datashade.R' 'debug.R' 'device.R' 'extendr-wrappers.R' 'paint.R' 'scene-model.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 2026-07-12 18:00:32 UTC

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

RemoteRef HEAD

RemoteSha 71266cc1a8481204e00ae23800b5595785e2aba3

Contents

`as_mask` *Masks*

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_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

x	A grid grob/gTree/gtable, or a ggplot object.
width, height	Page size in inches.
dpi, bg	As in vl_scene().
path	Output file (.png/.svg/.pdf).
text	Passed to render() (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 *Coerce an object to a vellum scene*

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

`x`	An object to coerce: a `vellum_scene`, or a type a downstream package has taught to compile by defining an `as_vellum_scene()` method.
`...`	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,
  colors = c("#deebf7", "#08306b"),
  how = c("eq_hist", "log", "cbrt", "linear"),
  interpolate = FALSE,
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)
```

Arguments

`x`, `y`	Point coordinates (plain numerics, in data space).
`weight`	Optional per-point weight; cells accumulate the summed weight instead of a plain count. `NULL` counts. A scalar is recycled to every point; otherwise `weight` must be the same length as `x`.
`width`, `height`	Aggregation grid size in cells (= output raster pixels).
`xlim`, `ylim`	Data range to bin over; default the finite range of `x`/`y`.
`colors`	Two or more colours forming the low-to-high density ramp.
`how`	Density-to-colour mapping: `"eq_hist"` (histogram equalisation — datashader's default, reveals structure across orders of magnitude), `"log"`, `"cbrt"` (cube root), or `"linear"`.
`interpolate`	Passed to `raster_grob()`; `FALSE` keeps hard bin edges.
`name`, `vp`, `id`, `role`	Passed to `raster_grob()` (see grob).

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()`.

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)
```

`describe` *Set a scene's accessibility name and description*

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

`scene`	A `vl_scene()`.
`title`	An accessible name (short), or `NULL` to leave unset.
`desc`	A long description / alt text, or `NULL` 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` *Display a scene in the active graphics device*

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)
```

`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"
)

radial_gradient(
  colours,
  stops = NULL,
  cx = 0.5,
  cy = 0.5,
  r = 0.5,
  units = "npc",
  extend = "pad"
)
```

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"`.
`cx`, `cy`, `r`	Centre and radius of a radial gradient (default centred, radius `0.5` npc).

Value

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

Examples

```
linear_gradient(c("white", "navy"))
radial_gradient(c("yellow", "red"), cx = 0.5, cy = 0.5, r = 0.5)
```

`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
```

```
)

lines_grob(
  x,
  y,
  arrow = NULL,
  start_cap = NULL,
  end_cap = NULL,
  offset = NULL,
  sketch = NULL,
  gp = vl_gpar(),
  name = NULL,
  vp = NULL,
  id = NULL,
  role = NULL
)

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

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

Arguments

x, y	Coordinates (`vl_unit()` or numeric).

`width`, `height` Grob size (`vl_unit()` or numeric), recycled like `x`/`y`. For most grobs the drawn rectangle size. For `hexagon_grob()`, the optional per-hexagon **full** corner-to-corner extent along x/y: when both are given they override `size` (resolved per-axis, so a hexagon can be *non-regular* and tile a non-square lattice — e.g. `"native"` units tile in data space regardless of device aspect); a *regular* flat hexagon is `height == width * sqrt(3) / 2`; leave both `NULL` to use circumradius `size`; must be given together. For `loop_grob()`, `width` is instead a dimensionless lateral petal scale in `(0, 1]` (recycled per loop): `1` (default) is the full teardrop, smaller narrows the petal's **waist** without shortening it (the igraph "narrowing" factor).

`sketch` Optional `sketch()` spec for a hand-drawn look; `NULL` = crisp.

`gp` Graphical parameters, from `vl_gpar()`.

`name` Optional name (for `edit_node()`).

`vp` Optional `vl_viewport()` to draw this grob inside.

`id` For most grobs, an optional semantic identifier emitted by the SVG backend as `data-vellum-id` (for interactivity, accessibility, and testing; ignored by raster/PDF). **For `path_grob` only**, `id` instead groups points (one value per point) into closed sub-paths: all points sharing an `id` form one sub-path (so a hole is a separate `id`), in first-appearance order (à la grid); `NULL` makes a single sub-path.

`role` Optional ARIA role, emitted by the SVG backend as `role=` for accessibility (ignored by the raster and PDF backends).

`key` Optional per-element data key(s) for the batched marks (`points_grob`, `circle_grob`, `rect_grob`, `segments_grob`, `hexagon_grob`, `sector_grob`), recycled to the element count like `fill`. Emitted by the SVG backend as `data-key` on each element and surfaced by `scene_model()` — 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.

`meta` 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 `scene_model()`. `NULL` (default) = none.

`r` Radius (`vl_unit()` or numeric).

`arrow` An `vl_arrow()` spec to draw heads on the line/segment ends, or `NULL` for none.

`start_cap`, `end_cap` Optional **absolute-length** `vl_unit()`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 `segments_grob()` the caps are per-element (scalar or length-n, recycled like the coordinates); for `lines_grob()` a single (scalar) cap trims each end of the whole polyline. `NULL` (default) leaves the endpoint untouched. When an `vl_arrow()` is also present its head is placed at the *capped* 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 >= the segment length draws nothing; a zero-length segment is skipped).

`offset` Optional **absolute-length** `vl_unit()` (`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 `segments_grob()` 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 `lines_grob()` a single (scalar) offset rigidly translates the whole polyline along the perpendicular of its overall direction. Applied **before** `start_cap`/`end_cap` and the arrowhead (offset, then cap, then head). `NULL`/`0` (default) leaves the geometry untouched.

`n` Number of points to sample the curve at (flattened to a polyline).

`shape` Marker shape(s): `"circle"` (default), `"square"`, `"triangle"`, `"diamond"`, `"plus"`, or `"cross"`, recycled per point. Filled shapes use `gp$fill` (and outline `gp$col`); `"plus"`/`"cross"` are stroke-only.

`open` If `FALSE`, the spline is closed (wraps end to start).

`size` Loop extent: an **absolute** `vl_unit()` (`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 `size` (same `x`/`y`/`angle`) for concentric teardrops. Recycled per loop.

`fill` Per-element fill colour(s), recycled to the number of sectors. `NULL` falls back to `gp$fill`.

`orientation` Hexagon orientation: `"flat"` (default, flat top/bottom edge) or `"pointy"` (vertex at top). `size` is the circumradius (centre to vertex).

`r0`, `r1` Inner and outer radius of each sector (`vl_unit()` or numeric; numeric is treated as `"native"`). `r0 = 0` gives a pie slice; `r0 == r1` gives an arc outline (stroke only, no fill).

`theta0`, `theta1` Start and end angle of each sector, in **radians**, with 0 at 3 o'clock and increasing counter-clockwise.

`foot` Node radius the loop's two **feet** attach at (an **absolute** `vl_unit()`; `0` = both feet at the vertex, like igraph). A positive `foot` 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.

`angle` Outward direction of the loop in **radians** (which way the teardrop bulges away from the vertex, e.g. away from the layout centroid).

`x0`, `y0`, `x1`, `y1` Segment start/end coordinates (`vl_unit()` or numeric).

`rule` Fill rule: `"winding"` (non-zero, default) or `"evenodd"`.

`image` A raster image: a `grDevices::as.raster()`-compatible object — a matrix/array of colours or greyscale values, or a `raster` object.

`interpolate` Smoothly interpolate when scaling (default `TRUE`)? `FALSE` keeps hard pixel edges.

label	Character string(s) to draw.
just	Justification: c(hjust, vjust) as names ("left", "centre", "right", "bottom", "top") or numbers in [0, 1].
rot	Rotation in degrees, counter-clockwise.

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

grob	A grob (or composite subtree) to measure.
mult	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 picking primitive the retained scene graph enables (base grid offers only `grid.locator()`). The scene is compiled into a colour pick-buffer (each grob drawn in a colour encoding its id, respecting clipping and paint order), so the result is geometry-, clip- and overlap-exact. Markers and text are matched by their bounding box; lines and segments by a small pick band.

Usage

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

Arguments

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

`md` *Rich-text labels (markdown subset)*

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

`text`	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. `**a^2^**`). `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 *Inspect and edit a scene by node name*

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

Value

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

scene_model *A serializable, per-element model of a scene*

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_raster *Read a rendered scene back as pixels*

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)
```

Arguments

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

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_svg *Render a scene to an SVG string*

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 `vl_scene()`.
`text`	For SVG output, how text is written: `"native"` (default) emits selectable `<text>` 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_model()`

sketch *Hand-drawn ("sketch") rendering*

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

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

`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,
  lineheight = NULL,
  name = NULL
)
```

Arguments

`col`	Stroke/text colour.
`fill`	Fill colour, or a gradient from `linear_gradient()` / `radial_gradient()`.
`lwd`	Line width (1 == 1/96 inch).
`alpha`	Opacity multiplier in `[0, 1]`.
`lty`	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 `lwd`.
`lineend`	Line cap: `"round"` (default), `"butt"`, or `"square"`.
`linejoin`	Line join: `"round"` (default), `"mitre"`, or `"bevel"`.
`linemitre`	Mitre limit (>= 1) for mitre joins; default 10.
`fontfamily`	Font family (text grobs).
`fontface`	One of `"plain"`, `"bold"`, `"italic"`, `"bold.italic"`.
`fontsize`	Font size in points.
`lineheight`	Line-height multiple.
`name`	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)
```

---

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 `vl_unit()` (default `vl_unit(0.25, "in")`).
`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"))
```

`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_gpar` *Graphical parameters*

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,
  lineheight = NULL
)
```

Arguments

`col`	Stroke/text colour.
`fill`	Fill colour, or a gradient from `linear_gradient()` / `radial_gradient()`.
`lwd`	Line width (1 == 1/96 inch).
`alpha`	Opacity multiplier in `[0, 1]`.
`lty`	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 `lwd`.
`lineend`	Line cap: `"round"` (default), `"butt"`, or `"square"`.
`linejoin`	Line join: `"round"` (default), `"mitre"`, or `"bevel"`.
`linemitre`	Mitre limit (>= 1) for mitre joins; default 10.
`fontfamily`	Font family (text grobs).
`fontface`	One of `"plain"`, `"bold"`, `"italic"`, `"bold.italic"`.
`fontsize`	Font size in points.
`lineheight`	Line-height multiple.

Value

A `gpar` object.

Examples

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

`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

`grob`	A grob, or a list of grobs, drawn into the tile (their `0..1` npc coordinates map to the tile, painted in order).
`width`, `height`	Size of one tile cell (default `0.1` npc).
`x`, `y`	Cell centre (default centred).
`units`	Coordinate system for the geometry; see `linear_gradient()`.
`extend`	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: PNG raster, SVG `<image>` in a `<pattern>`. The PDF backend has no image support yet, so a pattern degrades to the tile's average colour there.

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_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)
```

Arguments

`width`, `height`	Page size (`vl_unit()` or numeric inches).
`dpi`	Resolution in dots per inch.
`bg`	Background colour (or `NA` for transparent).
`gp`	Page-level graphical parameters (`vl_gpar()`) carried by the root viewport; inherited by everything drawn (e.g. a default `col`/`fontsize`).
`xscale`, `yscale`	Native coordinate range of the root viewport, so `"native"` units work at the page level without an explicit `push()`.
`clip`	Clip drawing to the page rectangle?
`title`, `desc`	Accessibility: an accessible **name** (a short title) and a longer **description** (alt text) for the scene. When either is set, the SVG backend emits `role="img"` + `<title>`/`<desc>` (referenced by `aria-labelledby`) 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.
`scene`	A `vl_scene()`.
`vp`	A `vl_viewport()`.
`grob`	A grob (see grob).
`n`	Number of viewport levels to ascend.
`path`	Output file path; the format is taken from the extension (`.png`, `.svg`, or `.pdf`).

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

`debug` 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`.

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_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"
)

vl_strheight(
  label,
  family = "",
  fontface = "plain",
```

```
  fontsize = 12,
  cex = 1,
  unit = "in"
)
```

Arguments

label	Character vector of strings to measure.
family	Font family (e.g. "sans", "serif", "mono", or a specific family name). "" 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".

Value

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

Examples

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

---

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

values	Numeric vector of magnitudes.
units	Character vector of unit names, recycled against values.
data	Optional list supplying context for derived units: label, fontfamily, fontface, fontsize, lineheight.
x	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,
  name = NULL,
  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).
width, height	Size (vl_unit() or numeric, in the parent).
xscale, yscale	Length-2 native coordinate ranges.
angle	Rotation in degrees, counter-clockwise about the centre.
clip	Clip drawing to this viewport: TRUE/FALSE for the viewport rectangle, or a polygon_grob()/path_grob() (in this viewport's coordinates) to clip to an arbitrary path.
gp	Inheritable graphical parameters, from vl_gpar().
layout	An optional grid_layout().
row, col	Cell (1-based) of the parent's layout to place into.
rowspan, colspan	Number of cells to span.
mask	An optional mask: a grob (or list of grobs), or an as_mask() result. The viewport's contents are rendered as an isolated layer and the mask modulates their visibility.

`alpha` Optional group opacity in `[0, 1]`. The viewport's contents are composited as a single isolated layer at this opacity, so overlapping elements do not accumulate (unlike per-element `vl_gpar(alpha=)`). `NULL` (default) means fully opaque.

`blend` Optional blend mode for compositing the viewport's contents (as an isolated layer) onto the backdrop below it. One of `"normal"` (default), `"multiply"`, `"screen"`, `"overlay"`, `"darken"`, `"lighten"`, `"color-dodge"`, `"color-burn"`, `"hard-light"`, `"soft-light"`, `"difference"`, `"exclusion"`, `"hue"`, `"saturation"`, `"color"`, or `"luminosity"` (the CSS `mix-blend-mode` set). `NULL`/`"normal"` is ordinary over-compositing.

`name` Optional name (for `edit_node()`).

`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

`scene`	A `vl_scene()` (or anything with an `as_vellum_scene()` method).
`name`	A node name (set via the `name` 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