

Package ‘scimesh’

September 22, 2026

Type Package

Title Headless Publication-Quality 3D Mesh Rendering Engine

Version 0.4.0

Description A fast, GPU-free 3D software renderer written in modern C++17 with native R bindings. Renders triangle meshes to publication-quality images entirely on the CPU, requiring no display server or graphics hardware. Features multi-light Blinn-Phong shading, screen-space ambient occlusion, anti-aliasing, depth fog, transparency, wireframe rendering, texture mapping, screen-space lines and text labels, and procedural geometry generation. Supports standard mesh file formats with PNG and PPM output. Works on high-performance computing clusters, headless servers, containers, and continuous integration pipelines, making it suitable for scientific visualization across neuro-imaging, molecular structures, and general 3D graphics.

License MIT + file LICENSE

URL <https://github.com/dfsp-spirit/scimesh>,
<https://dfsp-spirit.github.io/scimesh/>

BugReports <https://github.com/dfsp-spirit/scimesh/issues>

Imports Rcpp (>= 1.0.0)

LinkingTo Rcpp

Suggests testthat (>= 3.0.0), png, freesurferformats, viridisLite,
knitr, rmarkdown

VignetteBuilder knitr

SystemRequirements C++17

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation yes

Config/roxygen2/version 8.0.0

RoxygenNote 7.3.3

Author Tim Schäfer [aut, cre],
 Martin Hořňovský [ctb] (Author of Catch2
 (cpp_tests/catch_amalgamated.{h,cpp})),
 Christophe Riccio [ctb] (Author of GLM - OpenGL Mathematics
 (src/third_party/glm/)),
 Dimitri Diakopoulos [ctb] (Author of tinyply
 (src/third_party/tinyply.{h,cpp})),
 Tim Schäfer [ctb] (Author of libfs (src/third_party/libfs.h)),
 Sebastian Reiter [ctb] (Author of stl_reader
 (src/third_party/stl_reader.h)),
 Sean Barrett [ctb] (Author of stb libraries
 (src/third_party/stb_image*.h, src/third_party/stb_truetype.h)),
 Rasmus Andersson [ctb] (Author of the Inter font
 (inst/extdata/Inter-Regular.ttf, SIL OFL 1.1))

Maintainer Tim Schäfer <ts+code@rcmd.org>

Repository CRAN

Date/Publication 2026-09-22 20:40:02 UTC

Contents

apply_colormap	4
bezier_path	5
camera	6
camera_auto	7
camera_fit_scene	8
camera_orbit	9
clip_plane	10
colorbar_horizontal	11
colorbar_vertical	13
compose_layout	14
compute_vertex_normals	15
default_font	16
diverging_colormap	16
flip_uvs	17
font_info	18
generate_arrow	19
generate_axes	20
generate_bbox	20
generate_cone	21
generate_cuboid	22
generate_cylinder	22
generate_multi_cylinders	23
generate_multi_spheres	24
generate_plane	25
generate_pyramid	26
generate_sphere	27
generate_tetrahedron	27

generate_torus	28
generate_tube	29
generate_tubes	30
image_apply_contrast	31
image_crop	32
image_crop_to_content	32
image_grow	33
image_merge	34
image_rotate_90	35
image_scale	35
image_to_array	36
line_layer	36
mesh_bbox	38
mesh_from_rgl	39
mesh_to_rgl	39
path_curvature	40
path_length	41
read_obj	42
read_ply	42
read_stl	43
render_lines	44
render_mesh	45
render_options	46
render_points	49
render_scene	50
render_segments	51
render_spheres	53
render_text	54
render_triangles	55
resample_path	55
rotate_mesh	56
scale_mesh	57
scene	58
scene_set_line_affects_bounds	59
set_mesh_alpha	61
spline_path	62
stack_horizontal	63
stack_vertical	64
text_extent	65
text_layer	66
transform_mesh	68
translate_mesh	69
viridis_colormap	70
world_to_screen	70
write_gltf	71
write_png	72
write_stl	73
write_tga	74

apply_colormap	<i>Apply a colormap to numerical data</i>
----------------	---

Description

Maps numeric per-vertex (or per-element) data to RGBA colours using a colormap. Handles multi-dataset data (e.g., two brain hemispheres), NaN values, and optional outlier clipping via winsorizing.

Usage

```
apply_colormap(  
  data,  
  colormap = viridis_colormap(256L),  
  limits = NULL,  
  nan_color = c(0.5, 0.5, 0.5, 1),  
  winsor_percentiles = NULL  
)
```

Arguments

data	A numeric vector, or a list of numeric vectors for multi-dataset mapping (e.g., list(lh_data, rh_data)).
colormap	A colormap specification: a function f(n) returning n hex colour strings (e.g., viridis_colormap), a character vector of hex colours, or an Nx3/Nx4 matrix of RGBA values in [0,1]. Default is viridis_colormap(256L).
limits	How the data value range is determined. NULL (default) auto-detects from finite values after winsorizing. c(min, max) sets an explicit fixed range. "global" pools all datasets for a shared range. "each" uses independent per-dataset ranges.
nan_color	RGBA colour for NaN/NA values as a length-3 (RGB) or length-4 (RGBA) numeric vector in [0,1]. Default mid-grey.
winsor_percentiles	Optional c(lower, upper) percentiles for outlier clipping, e.g. c(0.02, 0.98). NULL disables winsorizing.

Value

If data is a single vector: an Nx4 numeric matrix of RGBA colours. If data is a list: a list of Nx4 matrices.

Attributes on the result provide metadata. Single-dataset: data_min, data_max, raw_min, raw_max, winsor_lo, winsor_hi, nan_count. Multi-dataset: pooled_data_min, pooled_data_max (use for a colourbar), data_ranges, winsor_cutoffs, nan_counts.

Examples

```
data <- c(1.2, 3.4, NA, 2.1, 5.0, 2.8)
colors <- apply_colormap(data)

noisy <- c(rnorm(95, mean = 50, sd = 10), 200, -50)
colors <- apply_colormap(noisy, winsor_percentiles = c(0.02, 0.98))

lh <- c(2.3, 2.1, NA, 3.4)
rh <- c(2.5, 2.0, NA, 3.1)

colors <- apply_colormap(list(lh, rh),
  colormap = viridis_colormap(256L),
  limits = "global",
  winsor_percentiles = c(0.02, 0.98),
  nan_color = c(1, 1, 1, 1))
```

bezier_path

*Sample a Bezier curve from a control polygon***Description**

Evaluates a Bezier curve of any degree with de Casteljau's algorithm. Unlike [spline_path](#), the control points are **not** points on the curve: only the first and the last one are, and the rest pull the curve like a magnet. This is the function to use when the input really is a control polygon (a font outline, a vector graphics path, a designed shape); use [spline_path](#) when your points are positions the curve should pass through.

Usage

```
bezier_path(control_points, samples = 64L)
```

Arguments

control_points Nx3 numeric matrix of control points (at least 2).

samples Number of points to emit along the curve (default 64, at least 2). Both endpoints are included in the result.

Value

An Nx3 numeric matrix of path points, or an empty (0-row) matrix if there are fewer than two control points.

See Also

[spline_path](#), [generate_tube](#)

Examples

```
# A quadratic Bezier arc, drawn as a tube:
arc <- bezier_path(matrix(c(0, 0, 0, 1, 1, 0, 2, 0, 0), ncol = 3, byrow = TRUE))
nrow(arc)
```

camera

Create a camera specification

Description

Defines a camera for rendering by specifying the eye position, look-at center, up vector, projection type, and field of view.

Usage

```
camera(
  eye,
  center,
  up = c(0, 1, 0),
  projection = c("perspective", "orthographic"),
  fov = 45
)
```

Arguments

eye	Numeric vector of length 3: camera position.
center	Numeric vector of length 3: point the camera looks at.
up	Numeric vector of length 3: camera up direction.
projection	Projection type: "perspective" (default) or "orthographic".
fov	Field of view in degrees (perspective only).

Value

A camera list suitable for `render_mesh()` or `render_scene()`.

Examples

```
cam <- camera(eye = c(0, 0, 5), center = c(0, 0, 0))
cam$eye
```

camera_auto

Auto-frame a camera to fit a mesh or vertex set

Description

Computes a camera position that frames the entire mesh in view. The camera is placed on the side given by `direction` and looks back at the mesh, at a distance that ensures the mesh fits within the field of view.

Usage

```
camera_auto(
  mesh,
  direction = c(0, 0, -1),
  up = c(0, 1, 0),
  fov = 45,
  margin = 1.1,
  rgl_compat = FALSE,
  projection = c("perspective", "orthographic")
)
```

Arguments

<code>mesh</code>	Either an Nx3 numeric matrix of vertex positions, or a mesh descriptor list with a <code>vertices</code> component.
<code>direction</code>	Direction from the mesh towards the camera, as a length-3 vector: it selects the side you view the mesh from (the camera is placed at center + <code>direction</code> * distance). For example, <code>c(0, 0, 1)</code> gives a front view of a mesh that faces +Z, and <code>c(1, 0, 0)</code> looks at it from its +X side. Ignored when <code>rgl_compat = TRUE</code> .
<code>up</code>	The up vector as a length-3 vector. Default <code>c(0, 1, 0)</code> . Ignored when <code>rgl_compat = TRUE</code> .
<code>fov</code>	Field of view in degrees. Default 45° (30° when <code>rgl_compat = TRUE</code>).
<code>margin</code>	Extra margin factor (1.0 = tight fit, 1.1 = 10% margin).
<code>rgl_compat</code>	Logical. If TRUE, use rgl's camera defaults and bounding-sphere distance formula. Default FALSE.
<code>projection</code>	Projection type: "perspective" (default) or "orthographic". When orthographic, the camera distance is computed to tightly frame the mesh regardless of FOV.

Details

When `rgl_compat = TRUE`, the camera mimics rgl's default auto-framing behaviour: a 30° FOV, 15° elevation, and the distance is computed from the *bounding sphere* of the mesh (the half-diagonal of the axis-aligned bounding box), reproducing the formula `distance = sphere_radius / sin(FOV/2)` used by rgl.

Value

A camera list, with S3 class "scimesh_camera".

Note

This function frames a mesh (or a set of vertices). It does not know about scene contents such as line layers or text labels; use [camera_fit_scene](#) to fit a camera to a whole scene, including the line layers that contribute to the scene bounds.

Examples

```
verts <- matrix(c(-1,-1,-1, 1,-1,-1, 1,1,-1, -1,1,-1,
                 -1,-1, 1, 1,-1, 1, 1,1, 1, -1,1, 1), ncol = 3, byrow = TRUE)
tris <- matrix(c(0L,3L,2L, 0L,2L,1L, 4L,5L,6L, 4L,6L,7L,
                0L,1L,5L, 0L,5L,4L, 2L,3L,7L, 2L,7L,6L,
                0L,4L,7L, 0L,7L,3L, 1L,2L,6L, 1L,6L,5L), ncol = 3, byrow = TRUE)
mesh <- list(vertices = verts, triangles = tris)
cam <- camera_auto(mesh, direction = c(1, 1, 1))
cam_rgl <- camera_auto(mesh, rgl_compat = TRUE)
```

camera_fit_scene	<i>Fit a camera to a whole scene</i>
------------------	--------------------------------------

Description

Computes a camera that frames the contents of a scene (see [scene](#)), i.e. its meshes together with the line layers that contribute to the scene bounds (see [line_layer](#), parameter `affects_bounds`). Use this instead of [camera_auto](#) when the camera has to consider something else than a mesh: a scene that contains only line layers (e.g. a tractogram or a connectome without a brain surface) is framed by those lines, and decorative layers (`affects_bounds = FALSE`) are ignored.

Usage

```
camera_fit_scene(
  scene,
  direction = c(0, 0, -1),
  up = c(0, 1, 0),
  fov = 45,
  margin = 1.1,
  projection = c("perspective", "orthographic")
)
```

Arguments

scene	A scene descriptor list, see scene() .
direction	Length-3 view direction, from the camera towards the scene (default $c(0, 0, -1)$, i.e. looking along -Z).
up	Length-3 up vector (default $c(0, 1, 0)$).
fov	Vertical field of view in degrees (default 45).
margin	Scale factor applied to the fitted distance; values above 1 leave a margin around the content (default 1.1).
projection	Projection type, "perspective" (default) or "orthographic".

Details

The camera is placed on the line from the center of the bounding box along `direction`, at a distance that makes the content fit into the field of view, exactly like [camera_auto](#) does it for a mesh.

Value

A camera list (see [camera\(\)](#)) with class "scimesh_camera".

See Also

[camera_auto](#) for meshes, [scene](#), [scene_set_line_affects_bounds](#)

Examples

```
# A scene without any mesh is framed by its lines.
line <- line_layer(matrix(c(0, 0, 0), ncol = 3), matrix(c(2, 0, 0), ncol = 3))
sc <- scene(list(), lines = line)
cam <- camera_fit_scene(sc, direction = c(0, 0, -1))
```

camera_orbit

Orbit a camera around an axis

Description

Rotates a camera's eye position and up vector around its center by a given angle about a rotation axis. Useful for generating turntable-style frame sequences.

Usage

```
camera_orbit(camera, axis = c(0, 0, 1), angle_degrees)
```

Arguments

camera	A camera list from camera() or camera_auto().
axis	Rotation axis as a length-3 vector. Default c(0, 0, 1) (Z axis).
angle_degrees	Rotation angle in degrees.

Value

A camera list with S3 class "scimesh_camera".

Examples

```
mesh <- generate_torus(c(0, 0, 0))
cam <- camera_auto(mesh, direction = c(1, 1, 1))
cam2 <- camera_orbit(cam, axis = c(0, 0, 1), angle_degrees = 90)
```

clip_plane	<i>Create a clip plane specification</i>
------------	--

Description

Defines a clipping plane for render_options(). Geometry on the negative side of the plane is removed, i.e. a point p is kept when dot(normal, p) + offset >= 0.

Usage

```
clip_plane(normal, offset = 0, space = c("world", "eye"))
```

Arguments

normal	Numeric vector of length 3: the plane normal. It points toward the side of the plane that is <i>kept</i> . It does not have to be unit-length; it is normalized internally and offset is always interpreted as a distance in world units.
offset	Numeric scalar: the signed distance of the plane from the origin along normal, in world units. For example offset = -d places the plane at distance d from the origin (in the direction of normal).
space	Character, either "world" (default) or "eye": "world" The plane is fixed in world coordinates and does not move when the camera moves. This is the convention used by rgl's clipplanes3d(), VTK/PyVista, ParaView and three.js. "eye" The plane is defined relative to the camera, i.e. offset is a distance from the camera, and the plane moves and rotates with it.

Details

By default the plane is defined in **world space**, so the cut is a fixed feature of the scene: it does not move when the camera is moved around, which is what you want for cross-sections and multi-view figures. Set `space = "eye"` for a camera-relative plane that travels with the camera (the classic OpenGL `glClipPlane` behaviour), e.g. for cutaway views.

Value

A list with components `normal`, `offset` and `space`, suitable for the `clip_planes` argument of `render_options()`.

See Also

[render_options](#)

Examples

```
# World space (default): keep the half of the scene with x <= 0.
# The cut stays at x = 0, no matter where the camera is placed.
clip_plane(normal = c(-1, 0, 0), offset = 0)

# Keep only the part with z >= -0.5 (remove everything below z = -0.5):
clip_plane(normal = c(0, 0, 1), offset = 0.5)

# Eye space: remove everything closer than 2 units to the camera
# (a camera-attached cutaway).
clip_plane(normal = c(0, 0, -1), offset = -2, space = "eye")

# A world-space cut through a cuboid; the cut stays at x = 0 for any camera
cuboid <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
cut_opts <- render_options(clip_planes = list(
  clip_plane(normal = c(-1, 0, 0), offset = 0)))

mesh_pixels <- function(img) sum(image_to_array(img)[, , 1] < 1)
cam <- camera(eye = c(3, 3, 3), center = c(0, 0, 0))

full <- render_mesh(cuboid$vertices, cuboid$triangles, camera = cam)
cut <- render_mesh(cuboid$vertices, cuboid$triangles, camera = cam,
  options = cut_opts)

mesh_pixels(cut) < mesh_pixels(full) # TRUE: half of the cuboid is gone
```

Description

Creates a horizontal colorbar as a 4-channel RGBA array. The color strip and optional tick labels are rendered to PNG using base R graphics (headless-safe). Returns a 3D array suitable for `image_to_array()` or direct composition.

Usage

```
colorbar_horizontal(
  colormap,
  n_colors = 256L,
  width = 600L,
  height = 80L,
  ticks = NULL,
  tick_labels = NULL,
  data_range = c(0, 1),
  label_cex = 1,
  title = NULL,
  background = c(1, 1, 1, 1)
)
```

Arguments

<code>colormap</code>	A vector of colors or a function returning colors (e.g. <code>grDevices::hcl.colors</code>).
<code>n_colors</code>	Number of discrete color segments in the gradient.
<code>width</code>	Output width in pixels.
<code>height</code>	Output height in pixels.
<code>ticks</code>	Numeric vector of tick positions in data units (matching <code>data_range</code>). If <code>NULL</code> , ticks are computed automatically via <code>pretty()</code> restricted to the data range.
<code>tick_labels</code>	Character vector of tick labels. If <code>NULL</code> , defaults to formatted tick values.
<code>data_range</code>	The data range that ticks are specified in. Defaults to <code>c(0, 1)</code> .
<code>label_cex</code>	Label size multiplier.
<code>title</code>	Optional title string drawn above the color strip (horizontal) or to the right (vertical).
<code>background</code>	Background RGBA color (0-1 scale).

Value

A 3D array of dimensions (height, width, 4) with values in `[0, 1]`.

Examples

```
cbar <- colorbar_horizontal(viridis_colormap, data_range = c(-2, 3),
  title = "Value")
dim(cbar) # height x width x 4
```

colorbar_vertical	<i>Generate a vertical colorbar image</i>
-------------------	---

Description

Creates a vertical colorbar as a 4-channel RGBA array.

Usage

```
colorbar_vertical(
  colormap,
  n_colors = 256L,
  width = 80L,
  height = 600L,
  ticks = NULL,
  tick_labels = NULL,
  data_range = c(0, 1),
  label_cex = 1,
  title = NULL,
  background = c(1, 1, 1, 1)
)
```

Arguments

colormap	A vector of colors or a function returning colors.
n_colors	Number of discrete color segments in the gradient.
width	Output width in pixels.
height	Output height in pixels.
ticks	Numeric vector of tick positions in data units. If NULL, ticks are computed automatically via <code>pretty()</code> restricted to the data range.
tick_labels	Character vector of tick labels.
data_range	The data range that ticks are specified in.
label_cex	Label size multiplier.
title	Optional title string drawn to the right of the color strip.
background	Background RGBA color (0-1 scale).

Value

A 3D array of dimensions (height, width, 4) with values in $[0, 1]$.

Examples

```
cbar <- colorbar_vertical(viridis_colormap, data_range = c(0, 100),
  title = "Count")
dim(cbar)
```

compose_layout

*Compose multiple images into a single figure***Description**

Arranges rendered images (from `render_mesh()` or `render_scene()`) into a grid layout and optionally appends a colorbar. All composition is done with pure R array operations.

Usage

```
compose_layout(
  images,
  nrow = NULL,
  ncol = NULL,
  colorbar = NULL,
  colorbar_height = 80L,
  colorbar_width = 80L,
  background = c(0, 0, 0, 0),
  colorbar_side = c("right", "left"),
  crop = FALSE
)
```

Arguments

<code>images</code>	A list of images, each a list with width, height, pixels as returned by <code>render_mesh()</code> .
<code>nrow</code>	Number of rows in the grid layout.
<code>ncol</code>	Number of columns in the grid layout. If both <code>nrow</code> and <code>ncol</code> are <code>NULL</code> , a square-ish layout is chosen automatically.
<code>colorbar</code>	Optional colorbar array (from <code>colorbar_horizontal()</code> or <code>colorbar_vertical()</code>). Placed below if horizontal, to the right if vertical.
<code>colorbar_height</code>	Height of the colorbar row in pixels. Only used when appending a horizontal colorbar.
<code>colorbar_width</code>	Width of the colorbar column in pixels. Only used when appending a vertical colorbar.
<code>background</code>	Background RGBA color for padding (0-1 scale).
<code>colorbar_side</code>	For vertical colorbars, whether to place the bar on the "right" (default) or "left" of the brain images. Ignored for horizontal colorbars.
<code>crop</code>	Logical. If <code>TRUE</code> , transparent borders are cropped individually and images are padded to per-row height and per-column width for a tight layout with minimal white space. Default is <code>FALSE</code> (images must be same size).

Value

A list with width, height, pixels suitable for `write_png()` or `image_to_array()`.

Examples

```
mesh1 <- generate_cuboid(c(-1.5, 0, 0), c(0.5, 0.5, 0.5), c(1, 0, 0, 1))
mesh2 <- generate_cuboid(c( 1.5, 0, 0), c(0.5, 0.5, 0.5), c(0, 0, 1, 1))
img1 <- render_mesh(mesh1$vertices, mesh1$triangles)
img2 <- render_mesh(mesh2$vertices, mesh2$triangles)
result <- compose_layout(list(img1, img2), nrow = 1L)
tmp_file <- tempfile(fileext = ".png")
write_png(result, tmp_file)
```

`compute_vertex_normals`*Compute per-vertex normals for a mesh*

Description

Computes smooth vertex normals by averaging face normals. Returns the same mesh with a normals component (Nx3 numeric matrix). Useful for imported meshes that lack pre-computed normals.

Usage

```
compute_vertex_normals(mesh)
```

Arguments

`mesh` A mesh descriptor list with vertices and triangles.

Value

The mesh with a normals component added.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
mesh <- compute_vertex_normals(mesh)
nrow(mesh$normals)
```

default_font	<i>Path of the font used for text labels</i>
--------------	--

Description

Text labels are drawn with a TrueType font that ships with the package (inst/extdata/Inter-Regular.ttf, SIL Open Font License 1.1), so labels look the same everywhere and no system font is required. This function returns the path of the font that will be used by default.

Usage

```
default_font()
```

Details

It can be overridden without touching any code by setting the SCIMESH_FONT environment variable to the path of another .ttf file, or per call by passing font_file to [text_layer\(\)](#), [text_extent\(\)](#) and friends.

Value

A character scalar: the path to an existing font file.

See Also

[font_info](#), [text_layer](#)

Examples

```
default_font()
```

diverging_colormap	<i>Custom diverging colormap for neuroimaging</i>
--------------------	---

Description

Returns a blue-white-red diverging colormap suitable for displaying signed morphometry data (e.g. cortical thickness Z-scores).

Usage

```
diverging_colormap(n)
```

Arguments

n Number of colors.

Value

A character vector of hex color strings.

Examples

```
cols <- diverging_colormap(256)
plot(1:256, pch = 15, col = cols, cex = 2, axes = FALSE, xlab = "", ylab = "")
```

 flip_uv

Flip the texture coordinates of a mesh vertically

Description

scimesh stores texture coordinates in **image space**, with $v = 0$ at the *top* edge of the texture image — the same rule as every other coordinate in scimesh ($c(0, 0)$ addresses the top-left pixel of the texture image, $c(1, 1)$ the bottom-right one). OBJ and PLY files, OpenGL, rgl and tools like Blender and MeshLab use the opposite convention ($v = 0$ at the bottom), so UVs taken from those sources have to be converted once; this function does that, instead of you having to rewrite the second column by hand.

Usage

```
flip_uv(mesh)
```

Arguments

mesh A mesh descriptor (scimesh or rgl format, see [as_scimesh_mesh\(\)](#)).

Details

Geometry, colors and normals are untouched. A mesh without texture coordinates is returned unchanged, so calling this is safe either way.

Value

The mesh with flipped UVs.

See Also

[render_mesh](#) (the uv and texture arguments)

Examples

```
quad <- list(vertices = matrix(c(-1, -1, 0, 1, -1, 0, 1, 1, 0,
                                -1, -1, 0, 1, 1, 0, -1, 1, 0),
                                ncol = 3, byrow = TRUE),
             triangles = matrix(c(1, 2, 3, 1, 3, 4), ncol = 3, byrow = TRUE),
             # UVs with v = 0 at the bottom (OBJ/OpenGL convention)
             uv = matrix(c(0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0),
                          ncol = 2, byrow = TRUE))

flipped <- flip_uv(quad)
flipped$uv[, 2] # v is now measured from the top of the texture
```

font_info

*Information about the font used for text labels***Description**

Reports which font file is used (and where it lives), its family name and its vertical metrics. Useful to check that the bundled font was found and to see what a custom .ttf file contains.

Usage

```
font_info(font_file = NULL, size = 18)
```

Arguments

font_file Path to a .ttf file, or NULL for the bundled font (see [default_font\(\)](#)).
size Text height in output pixels (default 18).

Value

A list with components family, path, size, ascent, descent and line_gap.

See Also

[default_font](#), [text_layer](#)

Examples

```
font_info()
```

generate_arrow	<i>Generate an arrow mesh</i>
----------------	-------------------------------

Description

Creates a 3D arrow from `from` to `to`, with a cylindrical shaft and a conical head.

Usage

```
generate_arrow(  
  from,  
  to,  
  shaft_radius = 0.1,  
  head_radius = 0.3,  
  head_length = 0.6,  
  segments = 32,  
  color = c(1, 1, 1, 1)  
)
```

Arguments

<code>from</code>	Length-3 start point.
<code>to</code>	Length-3 end point (tip of the arrowhead).
<code>shaft_radius</code>	Radius of the shaft cylinder.
<code>head_radius</code>	Radius at the base of the conical head.
<code>head_length</code>	Length of the arrowhead.
<code>segments</code>	Subdivision count (default 32).
<code>color</code>	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_arrow(c(0, 0, 0), c(0, 2, 0))  
nrow(mesh$vertices)
```

generate_axes	<i>Generate XYZ axis arrows as cylinder meshes</i>
---------------	--

Description

Creates three coloured arrow meshes (red X, green Y, blue Z) from a centre point.

Usage

```
generate_axes(center = c(0, 0, 0), size = 1, shaft_radius = 0.02)
```

Arguments

center	Length-3 vector: origin of the axes.
size	Length of each axis.
shaft_radius	Cylinder radius for axis shafts.

Value

A mesh descriptor list suitable for `render_mesh()` or inclusion in a scene list.

Examples

```
axes_mesh <- generate_axes(size = 2)
nrow(axes_mesh$vertices)
```

generate_bbox	<i>Generate a wireframe bounding box mesh</i>
---------------	---

Description

Creates 12 edge segments around an axis-aligned bounding box.

Usage

```
generate_bbox(bbox, color = c(0, 0, 0, 1), radius = 0.01)
```

Arguments

bbox	A bounding box list from <code>mesh_bbox()</code> , or a mesh descriptor (in which case <code>mesh_bbox()</code> is called).
color	RGBA colour for the edges (length 4, 0-1 scale).
radius	Cylinder radius for the edges.

Value

A mesh descriptor list suitable for `render_mesh()` or inclusion in a scene list.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 2, 3))
bbox_mesh <- generate_bbox(mesh)
nrow(bbox_mesh$vertices)
```

generate_cone	<i>Generate a cone mesh</i>
---------------	-----------------------------

Description

Creates a cone from base to tip with the given base radius, subdivided into segments around the axis. The base cap is included.

Usage

```
generate_cone(base, tip, radius = 0.5, segments = 32, color = c(1, 1, 1, 1))
```

Arguments

base	Length-3 vector: centre of the circular base.
tip	Length-3 vector: tip of the cone.
radius	Base radius.
segments	Subdivision count (default 32).
color	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_cone(c(0, -1, 0), c(0, 1, 0), radius = 0.8)
nrow(mesh$vertices)
```

generate_cuboid	<i>Generate a cuboid mesh</i>
-----------------	-------------------------------

Description

Creates an axis-aligned cuboid (box) centred at center with the given half-extents along each axis.

Usage

```
generate_cuboid(center, half_extents, color = c(0.7, 0.7, 0.7, 1))
```

Arguments

center	Length-3 vector: centre of the cuboid.
half_extents	Length-3 vector: half-width, half-height, half-depth.
color	Length-4 RGBA colour (0-1 scale).

Value

A mesh descriptor list.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 2, 0.5))
nrow(mesh$vertices)
nrow(mesh$triangles)
```

generate_cylinder	<i>Generate a cylinder mesh</i>
-------------------	---------------------------------

Description

Creates a cylinder from start to end with the given radius, subdivided into segments around the axis. Both end caps are included unless caps = FALSE is passed.

Usage

```
generate_cylinder(
  start,
  end,
  radius = 0.5,
  segments = 32,
  color = c(1, 1, 1, 1),
  caps = TRUE
)
```

Arguments

start	Length-3 vector: cylinder start point.
end	Length-3 vector: cylinder end point.
radius	Cylinder radius.
segments	Subdivision count (default 32).
color	Length-4 RGBA colour.
caps	Whether to close both ends with caps (default TRUE). Pass FALSE for an open tube, which roughly halves the number of vertices and triangles. Useful for edges whose ends are hidden by other geometry.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_cylinder(c(0, -1, 0), c(0, 1, 0), radius = 0.5)
nrow(mesh$vertices)
open <- generate_cylinder(c(0, -1, 0), c(0, 1, 0), radius = 0.5, caps = FALSE)
nrow(open$vertices)
```

generate_multi_cylinders

Generate multiple cylinders as a single mesh

Description

Batched variant of generate_cylinder(): all cylinders are generated into one mesh. This is the function to use for thousands of straight edges, e.g. the edges of a network graph or a connectome. Pass caps = FALSE to leave the ends open, which is usually what you want when the ends are hidden inside spherical nodes (and roughly halves the geometry).

Usage

```
generate_multi_cylinders(
  from,
  to,
  radii = 0.1,
  colors = c(1, 1, 1, 1),
  segments = 12L,
  caps = TRUE
)
```

Arguments

from	Nx3 numeric matrix of start points (or a length-3 vector).
to	Nx3 numeric matrix of end points (same number of rows as from).
radii	Numeric vector of radii (length 1, recycled; or one per cylinder).
colors	RGBA colour(s): a single vector applied to all cylinders, or an Nx4 numeric matrix (values in [0, 1], alpha optional).
segments	Subdivision count around the circumference (default 12).
caps	Whether to close both ends of every cylinder (default TRUE).

Value

A mesh descriptor list with vertices, triangles and colors.

See Also

[generate_cylinder](#), [generate_tubes](#)

Examples

```
from <- matrix(c(0, 0, 0, 1, 0, 0), ncol = 3, byrow = TRUE)
to   <- matrix(c(0, 3, 0, 1, 3, 0), ncol = 3, byrow = TRUE)
mesh <- generate_multi_cylinders(from, to, radii = 0.1,
                                colors = c(0.7, 0.7, 0.7, 1), caps = FALSE)
nrow(mesh$vertices) > 0
```

generate_multi_spheres

Generate multiple spheres as a single mesh

Description

Batched variant of generate_sphere(): all spheres are generated into one mesh with a single vertex/triangle array, which is much faster than generating and merging them one by one. This is the function to use for thousands of nodes, e.g. the nodes of a network graph or a point cloud. The returned mesh can be added to a scene and rendered with render_scene() or render_mesh().

Usage

```
generate_multi_spheres(
  centers,
  radii = 1,
  colors = c(1, 1, 1, 1),
  segments = 16L
)
```

Arguments

centers	Nx3 numeric matrix of sphere centres (or a length-3 vector for a single sphere).
radii	Numeric vector of radii (length 1, recycled; or one per sphere).
colors	RGBA colour(s): a single vector applied to all spheres, or an Nx4 numeric matrix (values in $[0, 1]$, alpha optional).
segments	Subdivision count per sphere (default 16).

Value

A mesh descriptor list with vertices, triangles and colors.

See Also

[generate_sphere](#), [generate_multi_cylinders](#)

Examples

```
centers <- matrix(c(0, 0, 0, 2, 0, 0), ncol = 3, byrow = TRUE)
mesh <- generate_multi_spheres(centers, radii = c(0.5, 0.3),
                              colors = c(1, 0, 0, 1), segments = 12)
nrow(mesh$vertices) > 0
```

generate_plane	<i>Generate a planar quad mesh</i>
----------------	------------------------------------

Description

Creates a flat rectangular plane centred at center and oriented perpendicular to normal.

Usage

```
generate_plane(
  center = c(0, 0, 0),
  normal = c(0, 1, 0),
  half_size_x = 1,
  half_size_y = 1,
  color = c(0.7, 0.7, 0.7, 1)
)
```

Arguments

center	Length-3 vector: centre of the plane.
normal	Length-3 vector: surface normal.
half_size_x	Half-extent along the first tangent axis.
half_size_y	Half-extent along the second tangent axis.
color	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_plane(c(0, 0, 0), normal = c(0, 1, 0),
                      half_size_x = 2, half_size_y = 1)
nrow(mesh$vertices)
```

generate_pyramid	<i>Generate a square pyramid mesh</i>
------------------	---------------------------------------

Description

Creates a pyramid with a square base centred at `base_center` in the XZ plane, with the apex above it along Y.

Usage

```
generate_pyramid(
  base_center,
  apex,
  half_width = 1,
  color = c(0.7, 0.7, 0.7, 1)
)
```

Arguments

<code>base_center</code>	Length-3 vector: centre of the square base.
<code>apex</code>	Length-3 vector: position of the tip.
<code>half_width</code>	Half-width of the square base.
<code>color</code>	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_pyramid(c(0, 0, 0), c(0, 2, 0), half_width = 1)
mesh$vertices
```

generate_sphere	<i>Generate a sphere mesh</i>
-----------------	-------------------------------

Description

Creates a UV sphere centred at center with the given radius. The sphere is subdivided into segments rings and segments per ring.

Usage

```
generate_sphere(center, radius = 1, segments = 32, color = c(1, 1, 1, 1))
```

Arguments

center	Length-3 vector: sphere centre.
radius	Sphere radius.
segments	Subdivision count (default 32).
color	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_sphere(c(0, 0, 0), radius = 1.5, segments = 32)
nrow(mesh$vertices)
```

generate_tetrahedron	<i>Generate a tetrahedron mesh</i>
----------------------	------------------------------------

Description

Creates a tetrahedron (triangular pyramid) from four arbitrary 3D points.

Usage

```
generate_tetrahedron(p0, p1, p2, p3, color = c(0.7, 0.7, 0.7, 1))
```

Arguments

p0	Length-3 vector: first vertex.
p1	Length-3 vector: second vertex.
p2	Length-3 vector: third vertex.
p3	Length-3 vector: fourth vertex.
color	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_tetrahedron(
  c(0, 0, 0), c(1, 0, 0),
  c(0.5, 1, 0), c(0.5, 0.5, 1))
nrow(mesh$vertices)
```

generate_torus

Generate a torus mesh

Description

Creates a torus (donut shape) centred at center, lying in the XZ plane.

Usage

```
generate_torus(
  center = c(0, 0, 0),
  major_radius = 1,
  minor_radius = 0.3,
  major_segments = 32,
  minor_segments = 16,
  color = c(0.7, 0.7, 0.7, 1)
)
```

Arguments

center	Length-3 vector: centre of the torus.
major_radius	Radius of the ring (tube path).
minor_radius	Radius of the tube cross-section.
major_segments	Number of segments around the ring.
minor_segments	Number of segments around the tube.
color	Length-4 RGBA colour.

Value

A mesh descriptor list.

Examples

```
mesh <- generate_torus(major_radius = 2, minor_radius = 0.5)
nrow(mesh$vertices)
```

generate_tube	<i>Generate a tube (generalized cylinder) along a path</i>
---------------	--

Description

Sweeps a circular cross-section along the points of path, which allows for curved shapes such as arcs, Bezier samples of network edges or streamlines. A path of exactly two points produces the same mesh as `generate_cylinder()`.

Usage

```
generate_tube(  
    path,  
    radius = 0.1,  
    segments = 12L,  
    color = c(1, 1, 1, 1),  
    cap_start = TRUE,  
    cap_end = TRUE  
)
```

Arguments

path	Nx3 numeric matrix of path points (or a length-3 vector).
radius	Tube radius (default 0.1).
segments	Subdivision count around the circumference (default 12).
color	Length-4 RGBA colour.
cap_start	Whether to close the beginning of the tube (default TRUE).
cap_end	Whether to close the end of the tube (default TRUE).

Details

The cross-section frames are computed by parallel transport (rotation-minimizing frames), so the tube does not twist around its own axis. Consecutive duplicate points are removed; a path with fewer than two distinct points yields an empty mesh.

Value

A mesh descriptor list.

See Also

[generate_tubes](#), [generate_cylinder](#)

Examples

```
path <- matrix(c(0, 0, 0, 1, 1, 0, 2, 0, 0), ncol = 3, byrow = TRUE)
arc <- generate_tube(path, radius = 0.1, segments = 12,
                    cap_start = FALSE, cap_end = FALSE)
nrow(arc$vertices) > 0
```

generate_tubes

*Generate multiple tubes as a single mesh***Description**

Batched variant of `generate_tube()`: all tubes are generated into one mesh. Paths may differ in length. This is the function to use for curved edges, e.g. connectome edges drawn as arcs.

Usage

```
generate_tubes(
  paths,
  radii = 0.1,
  colors = c(1, 1, 1, 1),
  segments = 12L,
  caps = FALSE
)
```

Arguments

<code>paths</code>	List of Nx3 numeric matrices (one per tube). Each path needs at least two distinct points to produce geometry.
<code>radii</code>	Numeric vector of radii (length 1, recycled; or one per tube).
<code>colors</code>	RGBA colour(s): a single vector applied to all tubes, or an Nx4 numeric matrix (values in $[0, 1]$, alpha optional).
<code>segments</code>	Subdivision count around the circumference (default 12).
<code>caps</code>	Whether to close both ends of every tube (default FALSE, since batched tubes are typically connected at their ends).

Value

A mesh descriptor list.

See Also

[generate_tube](#), [generate_multi_cylinders](#)

Examples

```
paths <- list(matrix(c(0, 0, 0, 1, 1, 0), ncol = 3, byrow = TRUE),
               matrix(c(0, 0, 2, 1, 1, 2, 2, 0, 2), ncol = 3, byrow = TRUE))
mesh <- generate_tubes(paths, radii = 0.05, segments = 8)
nrow(mesh$vertices) > 0
```

image_apply_contrast *Apply contrast adjustment to an image*

Description

Applies a contrast stretch (S-curve) to the RGB channels of a rendered image. Formula: $(\text{value} - 0.5) * \text{contrast} + 0.5$, clamped to $[0, 1]$. The default 1.0 means no change. Values > 1.0 produce darker darks and lighter highlights.

Usage

```
image_apply_contrast(image, contrast = 1)
```

Arguments

image	An image list returned by <code>render_mesh()</code> or <code>render_scene()</code> .
contrast	Contrast multiplier. Default 1.0 (no change). Typical values: 1.1–1.2 for subtle S-curve, 1.5 for strong.

Value

A new image list with contrast-adjusted pixel data.

Examples

```
cube <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(cube$vertices, cube$triangles)
img <- image_apply_contrast(img, contrast = 1.1)
```

image_crop	<i>Crop an image to a rectangular region</i>
------------	--

Description

Crop an image to a rectangular region

Usage

```
image_crop(image, x, y, w, h)
```

Arguments

image	An image list returned by <code>render_mesh()</code> or similar.
x	Left edge of the crop region (0-based pixel coordinate).
y	Top edge of the crop region (0-based pixel coordinate).
w	Crop width in pixels.
h	Crop height in pixels.

Value

A new image list with the cropped dimensions.

Examples

```
cube <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(cube$vertices, cube$triangles)
img <- image_crop(img, 100, 50, 400, 300)
```

image_crop_to_content	<i>Crop an image to its content bounding box</i>
-----------------------	--

Description

Removes background-coloured margin from the specified edges of the image. The first non-background pixel found on each edge defines the crop boundary.

Usage

```
image_crop_to_content(image, direction, background)
```

Arguments

image	An image list.
direction	One of "left", "right", "horizontal" (both left and right), "top", "bottom", "vertical" (both top and bottom), or "all" (all four sides).
background	Numeric vector of length 4 with RGBA values in $[0, 1]$ defining the background colour to crop away.

Value

A new image list with cropped dimensions.

Examples

```
cube <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(cube$vertices, cube$triangles,
  options = render_options(background_color = c(0, 0, 0, 0)))
img <- image_crop_to_content(img, "all", c(0, 0, 0, 0))
```

image_grow

Grow an image by adding padding

Description

Expands the canvas by adding pixel rows/columns filled with a background colour.

Usage

```
image_grow(image, top, bottom, left, right, background)
```

Arguments

image	An image list.
top	Number of pixel rows to add above.
bottom	Number of pixel rows to add below.
left	Number of pixel columns to add to the left.
right	Number of pixel columns to add to the right.
background	Numeric vector of length 4 with RGBA values in $[0, 1]$.

Value

A new image list with the expanded dimensions.

Examples

```
cube <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(cube$vertices, cube$triangles)
img <- image_grow(img, 10, 10, 20, 20, c(1, 1, 1, 1))
```

image_merge	<i>Merge two images side by side or stacked</i>
-------------	---

Description

Merges another image into this one at the specified edge. For left/right merging, the heights must match. For top/bottom, the widths must match.

Usage

```
image_merge(image, other, direction)
```

Arguments

image	An image list.
other	Another image list.
direction	One of "left", "right", "top", "bottom".

Value

A new image list with the merged dimensions.

Examples

```
cube <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
sphere <- generate_sphere(c(0, 0, 0), radius = 0.5)
left <- render_mesh(sphere$vertices, sphere$triangles)
right <- render_mesh(cube$vertices, cube$triangles)
merged <- image_merge(left, right, "right")
```

image_rotate_90	<i>Rotate an image by 90 degrees</i>
-----------------	--------------------------------------

Description

Rotate an image by 90 degrees

Usage

```
image_rotate_90(image, clockwise = TRUE)
```

Arguments

image	An image list.
clockwise	Logical, if TRUE (default) rotates clockwise, otherwise counter-clockwise.

Value

A new image list with width and height swapped.

image_scale	<i>Scale an image (nearest-neighbour)</i>
-------------	---

Description

Resizes the image to the given dimensions using nearest-neighbour interpolation.

Usage

```
image_scale(image, new_width, new_height)
```

Arguments

image	An image list.
new_width	Target width in pixels.
new_height	Target height in pixels.

Value

A new image list with the new dimensions.

image_to_array	<i>Convert a rendered image to an RGBA array</i>
----------------	--

Description

Converts the output of `render_mesh()` or `render_scene()` into a 3-dimensional R array of dimensions (height x width x 4) with RGBA channels.

Usage

```
image_to_array(image)
```

Arguments

`image` An image list returned by `render_mesh()` or `render_scene()`.

Value

A 3D array of dimensions (height, width, 4) with values in $[0, 1]$.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(mesh$vertices, mesh$triangles)
arr <- image_to_array(img)
dim(arr) # height x width x 4
```

line_layer	<i>Create a line layer (screen-space lines, no geometry)</i>
------------	--

Description

Bundles a set of independent line segments with a width measured in pixels into a layer that can be added to a scene (see the `lines` argument of `scene()`), or drawn directly with `render_segments`. In contrast to tube meshes (`generate_tubes`), no geometry is created: the renderer draws the segments itself, so thousands of lines cost almost nothing, and a width of 1 stays 1 pixel wide no matter how far away the geometry is. This is what hardware line rendering (and `rgl::segments3d()`) does.

Usage

```

line_layer(
  from,
  to,
  colors = NULL,
  width = 1,
  depth_test = TRUE,
  lit = FALSE,
  affects_bounds = TRUE
)

```

Arguments

from	Nx3 numeric matrix of segment start points (or a length-3 vector for a single segment).
to	Nx3 numeric matrix of segment end points (same number of rows as from).
colors	RGBA colour(s): a single vector applied to all segments, or an Nx4 numeric matrix (values in $[0, 1]$, alpha optional). The default NULL uses the default <code>t_color</code> of the render options.
width	Line width in pixels (default 1).
depth_test	Whether to test the lines against the depth buffer (default TRUE). Set to FALSE to draw them on top of everything, which is only useful for opaque lines.
lit	Whether to apply lighting to the lines (default FALSE, i.e. a flat colour, like hardware-rendered lines).
affects_bounds	Whether this layer contributes to the bounding box of the scene, and thus to the camera fitted to it (default TRUE, see the description). Set to FALSE for decorative lines. The flag of a layer that is already part of a scene can be changed with scene_set_line_affects_bounds .

Details

Line layers are drawn together with the meshes of the scene, after them and against the same depth buffer, so opaque meshes can hide lines and opaque lines can hide meshes. Segments whose colors have an alpha value below 1 are drawn in the blended pass, back to front, exactly like translucent triangles.

Lines usually *are* the content of a figure (graph or connectome edges, streamlines, trajectories), so by default a layer contributes to the bounding box of its scene, exactly like a mesh does: it defines the extent that the camera has to cover. Set `affects_bounds = FALSE` for a layer that is decoration rather than content (a leader line to a label, an axis cross, a scale bar drawn as segments), so that it can never push the camera away from the data. A scene that contains no mesh at all is framed by its line layers even when they all opted out, since there would otherwise be no geometry to derive a camera from.

Value

A line layer object (a list with class `scimesh_lines`) for use in [scene\(\)](#) or [render_segments](#).

See Also

[render_segments](#), [generate_tubes](#)

Examples

```
from <- matrix(c(-1, 0, 0, 0, -1, 0), ncol = 3, byrow = TRUE)
to   <- matrix(c(1, 0, 0, 0, 1, 0), ncol = 3, byrow = TRUE)
layer <- line_layer(from, to, colors = c(0.2, 0.2, 0.2, 0.8), width = 2)
sc <- scene(list(generate_sphere(c(0, 0, 0), 0.5)), lines = layer)
```

mesh_bbox

Compute the axis-aligned bounding box of a mesh

Description

Compute the axis-aligned bounding box of a mesh

Usage

```
mesh_bbox(mesh)
```

Arguments

mesh A mesh descriptor list with vertices.

Value

A list with min and max (each length-3 numeric).

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 2, 3))
bb <- mesh_bbox(mesh)
bb$min
bb$max
```

mesh_from_rgl	<i>Convert an rgl tmesh3d to scimesh mesh format</i>
---------------	--

Description

Extracts vertices and triangle indices from an rgl tmesh3d object into the format expected by render_mesh(). Does not require the rgl package – any list with components vb (4xN homogeneous coordinates) and it (3xM index matrix) works.

Usage

```
mesh_from_rgl(tmesh)
```

Arguments

tmesh A list with components vb and it, as produced by rgl::tmesh3d().

Value

A mesh descriptor list with vertices (Nx3) and triangles (Mx3, 1-based indices).

Examples

```
fake <- list(vb = rbind(0:3, 0:3, 0:3, rep(1, 4)),
             it = matrix(1:6, nrow = 3))
m <- mesh_from_rgl(fake)
m$vertices
m$triangles
```

mesh_to_rgl	<i>Convert a scimesh mesh to rgl tmesh3d format</i>
-------------	---

Description

Builds an rgl-compatible triangular mesh from a scimesh mesh descriptor so that the result can be used with rgl::shade3d() or other rgl functions.

Usage

```
mesh_to_rgl(mesh, color = NULL, face_color = NULL)
```

Arguments

mesh	A scimesh mesh descriptor list with vertices (Nx3 matrix) and triangles (Mx3 integer matrix, 1-based).
color	Optional per-vertex colour, either a single length-4 RGBA vector (applied to all vertices) or an Nx4 matrix.
face_color	Optional per-face colour (Mx4 matrix).

Value

A list with components vb (4xN homogeneous coordinates), it (3xM 1-based index matrix), and optionally normals and mat (material), suitable for use with rgl's tmesh3d() and shade3d().

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
rgl_mesh <- mesh_to_rgl(mesh)
str(rgl_mesh)
```

path_curvature	<i>Curvature along a path</i>
----------------	-------------------------------

Description

Estimates the curvature at every point from the neighbouring points, using the standard formula $\kappa = |x' \times x''|/|x'|^3$, which does not care how the points are spaced. On a circle of radius r the values come out as $1/r$, and on a straight line as 0.

Usage

```
path_curvature(path, closed = FALSE)
```

Arguments

path	Nx3 numeric matrix of path points (or a length-3 vector).
closed	Whether the path loops back to its first point (default FALSE).

Details

This is a diagnostic tool rather than a rendering input: sweeping a tube of radius r along a curve whose curvature reaches κ folds the tube inside out, so a path can be swept safely up to a radius of $1/\max(\kappa)$, which is worth checking for tight turns in measured data ($\max(\text{path_curvature}(\text{path}))$).

Value

A numeric vector with one curvature value per input point. The endpoints of an open path report the value of their only neighbour; fewer than 3 points yield a zero-length vector, since curvature is undefined there.

See Also[spline_path](#), [generate_tube](#)**Examples**

```
# Largest tube radius that does not fold this path onto itself:
path <- spline_path(matrix(c(0, 0, 0, 1, 1, 0, 2, 0, 0, 3, 1, 0),
                           ncol = 3, byrow = TRUE))
1 / max(path_curvature(path))
```

path_length	<i>Length of a path</i>
-------------	-------------------------

Description

Sums the distances between consecutive points, plus the closing distance from the last point back to the first for a closed path. Useful to pick a sensible step for [resample_path](#), or to derive a sampling density from the size of the data.

Usage

```
path_length(path, closed = FALSE)
```

Arguments

path Nx3 numeric matrix of path points (or a length-3 vector).

closed Whether the path loops back to its first point (default FALSE).

Value

A single number: the arc length of the path (0 for fewer than two points).

See Also[resample_path](#)**Examples**

```
path_length(matrix(c(0, 0, 0, 1, 0, 0, 1, 1, 0), ncol = 3, byrow = TRUE))
```

read_obj	<i>Read a Wavefront OBJ file</i>
----------	----------------------------------

Description

Reads the geometry (vertices and triangles) of a Wavefront OBJ file and returns a scimesh mesh descriptor list with vertices and triangles. Normals and texture coordinates in the file are ignored: call `compute_vertex_normals()` if you need normals, and assign uv yourself (see `render_mesh`) if you want to render the mesh with a texture.

Usage

```
read_obj(path)
```

Arguments

path	Path to the OBJ file.
------	-----------------------

Value

A mesh descriptor list with vertices and triangles.

Examples

```
## Not run:
mesh <- read_obj("model.obj")
nrow(mesh$vertices)

## End(Not run)
```

read_ply	<i>Read a Stanford PLY file</i>
----------	---------------------------------

Description

Reads a PLY file (ASCII or binary) with optional per-vertex colors and returns a scimesh mesh descriptor list with vertices, triangles, and optionally colors. Texture coordinates in the file are ignored.

Usage

```
read_ply(path)
```

Arguments

path	Path to the PLY file.
------	-----------------------

Value

A mesh descriptor list.

Examples

```
## Not run:  
mesh <- read_ply("model.ply")  
nrow(mesh$vertices)  
  
## End(Not run)
```

read_stl

Read an STL file

Description

Reads an ASCII or binary STL file and returns a scimesh mesh descriptor list with vertices, triangles, and normals.

Usage

```
read_stl(path)
```

Arguments

path Path to the STL file.

Value

A mesh descriptor list.

Examples

```
## Not run:  
mesh <- read_stl("model.stl")  
nrow(mesh$vertices)  
  
## End(Not run)
```

render_lines	<i>Render line segments as thin cylinders</i>
--------------	---

Description

Generates a merged cylinder mesh from start/end point pairs, radii, and colors, then renders it.

Usage

```
render_lines(
  from,
  to,
  radii = 0.1,
  colors,
  camera,
  options = render_options(),
  segments = 12L
)
```

Arguments

from	Nx3 numeric matrix of segment start points.
to	Nx3 numeric matrix of segment end points.
radii	Numeric vector of cylinder radii (length N, or 1 recycled to N).
colors	Nx4 numeric matrix of RGBA colours, or a single colour recycled to N.
camera	A camera list.
options	Render options.
segments	Number of sides around the cylinder (default 12).

Value

An image list.

Examples

```
from <- matrix(c(0, 0, 0, 1, 1, 1), ncol = 3, byrow = TRUE)
to <- matrix(c(3, 0, 0, 0, 3, 0), ncol = 3, byrow = TRUE)
img <- render_lines(from, to, radii = 0.05,
  colors = c(0, 0, 1, 1),
  camera = camera_auto(rbind(from, to)))
tmp_file <- tempfile(fileext = ".png")
write_png(img, tmp_file)
```

render_mesh

*Render a 3D mesh to an image***Description**

Renders a single mesh using the scimesh software renderer. The mesh can be specified either as separate vertices/triangles matrices, as a scimesh mesh descriptor list, or as an rgl tmesh3d-style list (with vb/it components). rgl meshes are transparently converted via [mesh_from_rgl\(\)](#).

Usage

```
render_mesh(
  vertices,
  triangles = NULL,
  colors = NULL,
  face_colors = NULL,
  normals = NULL,
  uv = NULL,
  texture = NULL,
  camera = NULL,
  options = render_options()
)
```

Arguments

vertices	Either an Nx3 numeric matrix of vertex positions, or a scimesh mesh descriptor list (with vertices and triangles components), or an rgl-style list (with vb and it components).
triangles	Mx3 integer matrix of triangle indices (1-based). Ignored when vertices is a list.
colors	Optional Nx4 numeric matrix of RGBA vertex colors (0-1). The fourth column is the alpha value; $\alpha < 1$ renders the mesh translucently (see the Transparency section below). Use face_colors (Mx4) for per-triangle colours instead.
face_colors	Optional Mx4 numeric matrix of per-face RGBA colors, one row per triangle. When present, all three vertices of a triangle use the same colour. Takes precedence over vertex colors.
normals	Optional Nx3 numeric matrix of vertex normals.
uv	Optional Nx2 numeric matrix of texture coordinates (0-1). scimesh uses image-space UVs: $v = 0$ is the <i>top</i> edge of the texture image, so $c(0, 0)$ addresses its top-left pixel. UVs from OBJ/PLY files, rgl or Blender use the opposite convention and must be converted with flip_uvs() first.
texture	Optional texture image as a 3D array (H x W x 3 or 4) with values in $[0, 1]$, e.g. from <code>png::readPNG()</code> . Row 1 of the array is the top row of the image, matching the UV convention above.

camera	A camera list from camera() or camera_auto().
options	A render options list from render_options().

Value

A list with components width, height, and pixels (raw vector of RGBA values).

Transparency

The fourth column of colors (and of face_colors) is the alpha value: values < 1 make the mesh translucent, and the renderer blends it with whatever is behind it automatically - there is no flag to set. Per-vertex alpha is interpolated across each triangle, so a smooth fade is possible, and alpha = 0 makes geometry invisible (holes). Translucent triangles are drawn back-to-front after the opaque geometry, so they are correctly hidden by opaque meshes in front of them. Use `set_mesh_alpha()` to set one alpha value for a whole mesh.

Examples

```
# Render a simple colored triangle
verts <- matrix(c(0, 0, 0, 1, 0, 0, 0.5, 1, 0), ncol = 3, byrow = TRUE)
tris <- matrix(1L, nrow = 1, ncol = 3)
cols <- matrix(c(1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1), ncol = 4, byrow = TRUE)
img <- render_mesh(verts, tris, colors = cols)
tmp_file <- tempfile(fileext = ".png")
write_png(img, tmp_file)

# Render from a mesh descriptor list (scimesh format)
mesh_desc <- list(vertices = verts, triangles = tris, colors = cols)
img <- render_mesh(mesh_desc)
```

render_options	<i>Create render options</i>
----------------	------------------------------

Description

Create render options

Usage

```
render_options(
  width = 800L,
  height = 600L,
  shading = c("smooth", "flat"),
  backface_culling = TRUE,
  background_color = c(1, 1, 1, 1),
  default_color = c(0.7, 0.7, 0.7, 1),
  invert_normals = FALSE,
```

```

wireframe = FALSE,
wireframe_color = c(0, 0, 0, 1),
projection = c("perspective", "orthographic"),
specular_color = c(0, 0, 0, 0),
shininess = 0,
lights = NULL,
ambient = 0.3,
contrast = 1,
fog_enabled = FALSE,
fog_start = 0,
fog_end = 1,
fog_color = c(0, 0, 0, 0),
fog_space = c("world", "ndc"),
threads = 0L,
clip_planes = NULL,
ssao_enabled = FALSE,
ssao_radius = 16,
ssao_intensity = 0.8,
aa_samples = NULL,
near_plane = 0.1,
far_plane = 10000
)

```

Arguments

width	Output image width in pixels.
height	Output image height in pixels.
shading	Shading mode: "smooth" or "flat".
backface_culling	Whether to cull back-facing triangles.
background_color	Background RGBA color as numeric vector of length 4 (values 0-1).
default_color	Default vertex color when no colors are provided.
invert_normals	Whether to invert surface normals.
wireframe	Whether to render in wireframe mode.
wireframe_color	RGBA color for wireframe edges (0-1 scale). Default c(0, 0, 0, 1) (black).
projection	Projection type: "perspective" (default) or "orthographic". Orthographic gives a parallel projection (no perspective foreshortening), matching rgl's view3d(fov=0) convention.
specular_color	Specular highlight color (0-1 scale). When shininess > 0, a Blinn-Phong highlight in this colour is added where the surface faces the camera. Default c(0, 0, 0, 0) (off).
shininess	Specular exponent controlling highlight sharpness. Higher values produce a tighter spot. Typical values: 32 (soft plastic), 64 (shiny), 128 (glass). Default 0 (off).

lights	A list of light descriptors, each a list with position (length-3 direction vector or point position), color (length-4 RGBA, 0-1 scale), intensity (numeric, default 1), and directional (logical, default TRUE). When empty or NULL, a single headlight at $c(0, 0, 1)$ is used (the original behaviour).
ambient	Ambient light contribution (0-1). Default 0.3.
contrast	Contrast adjustment applied after shading, before uint8_t conversion. Default 1.0 (no change). Values > 1.0 produce darker darks and lighter highlights (S-curve). Formula: $(value - 0.5) * contrast + 0.5$, clamped to $[0, 1]$.
fog_enabled	Enable depth cueing (fog). Default FALSE.
fog_start	Distance where fog begins, i.e. where objects start fading toward fog_color. Default 0.
fog_end	Distance where fog is fully opaque. Must be larger than fog_start. Default 1.
fog_color	RGBA fog colour (0-1 scale). Defaults to background_color.
fog_space	Character, either "world" (default) or "ndc": the space (and therefore the unit) of fog_start and fog_end. "world" Distances in world units from the camera, measured along the viewing direction. fog_start = 20 means "fog starts 20 world units in front of the camera". This is independent of near_plane/far_plane and of the projection type. "ndc" Normalized device depth, i.e. the raw depth-buffer values in $[-1, 1]$, where -1 is the near plane, 0 the middle of the depth range and +1 the far plane. This is the legacy behaviour; it depends on the near/far plane settings and is strongly non-linear for perspective cameras.
threads	Number of render threads. 0 = auto-detect (use all cores), 1 = single-threaded (deterministic). Default 0. Requires OpenMP at compile time.
clip_planes	A list of clip planes (see clip_plane), or NULL (default) for no clipping. Each plane removes the geometry on its negative side, i.e. a point p is kept when $\text{dot}(\text{normal}, p) + \text{offset} \geq 0$; several planes are combined with a logical AND. By default p is the world-space position, so the cut is fixed in the scene and does not move when the camera moves; use clip_plane(..., space = "eye") for a camera-relative cut. Note that clip planes only apply to mesh and triangle rendering; render_points() and render_spheres() ignore them.
ssao_enabled	Enable screen-space ambient occlusion. Default FALSE.
ssao_radius	Screen-space sample radius in pixels. Default 16.
ssao_intensity	Occlusion strength (0-1). Default 0.8.
aa_samples	Anti-aliasing supersampling factor. Renders internally at width * aa_samples x height * aa_samples, then downsamples to the requested size via box averaging. Use 1 (the default) for no AA, 2 for 2x2 SSAA, 4 for 4x4. If NULL, the global option scimesh.aa_samples is used (which defaults to 1). Set that option once per session to enable AA for all render calls, e.g. options(scimesh.aa_samples = 2). Note that AA increases render time and memory roughly with $aa_samples^2$, and that thin lines and points get smoother edges from it.
near_plane	Distance of the near clipping plane (default 0.1). Geometry closer to the camera is clipped away. Also defines the depth range together with far_plane, which matters when fog_space = "ndc".
far_plane	Distance of the far clipping plane (default 10000). Must be larger than near_plane.

Value

A render options list for use with `render_mesh()` or `render_scene()`.

Examples

```
# Default options
opts <- render_options()

# High-resolution with anti-aliasing and specular highlights
opts <- render_options(width = 1200, height = 900,
  aa_samples = 2L,
  specular_color = c(0.4, 0.4, 0.4, 1),
  shininess = 64)

# Wireframe with transparent background
opts <- render_options(wireframe = TRUE,
  wireframe_color = c(0, 0, 0, 1),
  background_color = c(0, 0, 0, 0))

# World-space clip plane: keep the half of the scene with x <= 0.
# The cut stays at x = 0, whatever the camera does.
opts <- render_options(clip_planes = list(
  clip_plane(normal = c(-1, 0, 0), offset = 0)))

# Eye-space clip plane: additionally remove everything closer than 2 units
# to the camera (camera-attached cutaway).
opts <- render_options(clip_planes = list(
  clip_plane(normal = c(-1, 0, 0), offset = 0),
  clip_plane(normal = c(0, 0, -1), offset = -2, space = "eye")))

# Fog in world units (default): fade from 20 to 60 units away from the
# camera
opts <- render_options(fog_enabled = TRUE, fog_start = 20, fog_end = 60,
  fog_color = c(0.9, 0.95, 1, 1))

# Legacy normalized-device-depth fog, for backwards compatibility
opts <- render_options(fog_enabled = TRUE, fog_space = "ndc",
  fog_start = 0.5, fog_end = 1)

# Enable 2x2 anti-aliasing for this session: affects all subsequent
# render calls that do not pass \code{aa_samples} explicitly.
old <- options(scimesh.aa_samples = 2L)
opts <- render_options()
opts$aa_samples
options(old)
```

Description

Renders points as fixed-size filled circles in screen space with depth testing. Unlike `render_spheres()`, point size is measured in pixels and does not change with camera distance.

Usage

```
render_points(
  positions,
  colors,
  radius = 3,
  camera = camera_auto(positions),
  options = render_options()
)
```

Arguments

<code>positions</code>	Nx3 numeric matrix of point positions.
<code>colors</code>	Nx4 numeric matrix of RGBA colours (0-1 scale).
<code>radius</code>	Point radius in pixels.
<code>camera</code>	A camera list.
<code>options</code>	Render options.

Value

An image list.

Examples

```
pts <- matrix(c(0, 1, 2, 0, 1, 2, 0, 0, 0),
  ncol = 3)
colors = matrix(c(0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1), ncol = 4)
img <- render_points(pts, colors = colors, radius = 5)
tmp_file <- tempfile(fileext = ".png")
write_png(img, tmp_file)
```

render_scene

Render multiple meshes to an image

Description

Renders a list of meshes as a single scene using the scimesh software renderer. Each element can be a scimesh mesh descriptor or an rgl-style mesh (with `vb/i t`); rgl meshes are transparently converted.

Usage

```
render_scene(meshes, camera = NULL, options = NULL)
```

Arguments

meshes	Either a scimesh_scene object (see scene()), a list of mesh descriptors, or a list of scene nodes. Each mesh descriptor is a list with components vertices (Nx3 matrix), triangles (Mx3 integer matrix), and optionally colors, face_colors, normals, and default_color. Elements may also be rgl-style lists (with vb and it), which are converted automatically. A scimesh_scene may hold line layers and text layers, which are drawn together with the meshes (see line_layer and text_layer).
camera	A camera list from camera() or camera_auto(). Ignored (falls back to the scene's camera) when meshes is a scimesh_scene and camera is NULL.
options	A render options list from render_options(). Defaults to the scene's options (if any) or render_options().

Value

A list with components width, height, and pixels (raw vector of RGBA values).

Examples

```
# Render two cubes side by side
cube1 <- generate_cuboid(c(-1.5, 0, 0), c(0.8, 0.8, 0.8), c(1, 0, 0, 1))
cube2 <- generate_cuboid(c( 1.5, 0, 0), c(0.8, 0.8, 0.8), c(0, 0, 1, 1))
cam <- camera_auto(list(cube1, cube2), direction = c(1, 1, 1))
img <- render_scene(list(cube1, cube2), cam,
  render_options(width = 800, height = 600, background_color = c(1, 1, 1, 1)))
tmp_file <- tempfile(fileext = ".png")
write_png(img, tmp_file)

# Render multiple meshes together
scimesh_cube <- generate_cuboid(c(-1, 0, 0), c(0.5, 0.5, 0.5))
sphere <- generate_sphere(c(1, 0, 0), radius = 0.5, color = c(0, 1, 0, 1))
cam <- camera_auto(list(scimesh_cube, sphere), direction = c(1, 1, 1))
img <- render_scene(list(scimesh_cube, sphere), cam,
  render_options(width = 400, height = 300, background_color = c(1, 1, 1, 1)))
```

render_segments	<i>Render line segments directly to an image</i>
-----------------	--

Description

Convenience wrapper around [line_layer](#) for the case where no meshes are involved: the segments are drawn (with a screen-space width) into an image and nothing else. To combine lines with meshes, add the layer to a scene instead and render that scene.

Usage

```
render_segments(  
  from,  
  to,  
  colors = NULL,  
  width = 1,  
  camera = NULL,  
  options = render_options(),  
  lit = FALSE  
)
```

Arguments

from	Nx3 numeric matrix of segment start points (or a length-3 vector).
to	Nx3 numeric matrix of segment end points (same number of rows as from).
colors	RGBA colour(s): a single vector applied to all segments, or an Nx4 numeric matrix. NULL (the default) uses the default <code>t_color</code> of the render options.
width	Line width in pixels (default 1).
camera	A camera list, e.g. from camera or camera_auto . Defaults to a camera framing the segments.
options	Render options, see render_options .
lit	Whether to apply lighting (default FALSE, flat colour).

Value

An image list, see [render_scene](#).

See Also

[line_layer](#), [render_points](#)

Examples

```
from <- matrix(c(-1, 0, 0, 0, -1, 0), ncol = 3, byrow = TRUE)  
to <- matrix(c(1, 0, 0, 0, 1, 0), ncol = 3, byrow = TRUE)  
img <- render_segments(from, to, colors = c(1, 0, 0, 1), width = 3)  
tmp_file <- tempfile(fileext = ".png")  
write_png(img, tmp_file)
```

render_spheres	<i>Render multiple spheres from point data</i>
----------------	--

Description

Generates a merged sphere mesh from a set of center points, radii, and colors, then renders it with the given camera and options.

Usage

```
render_spheres(
  centers,
  radii,
  colors,
  camera,
  options = render_options(),
  segments = 16L
)
```

Arguments

centers	Nx3 numeric matrix of sphere centre coordinates.
radii	Numeric vector of sphere radii (length N, or 1 recycled to N).
colors	Nx4 numeric matrix of RGBA colours (0-1 scale), or a single colour recycled to N.
camera	A camera list from camera() or camera_auto().
options	Render options from render_options().
segments	Number of latitude/longitude segments per sphere (default 16).

Value

An image list with width, height, pixels.

Examples

```
centers <- matrix(c(0, 2, 4, 0, 0, 0, 0, 0, 0), ncol = 3)
img <- render_spheres(centers, radii = 0.5,
                      colors = c(1, 0, 0, 1),
                      camera = camera_auto(centers))
tmp_file <- tempfile(fileext = ".png")
write_png(img, tmp_file)
```

render_text

*Render text labels to an image***Description**

Convenience wrapper around [text_layer](#) for the case where no meshes are involved: the labels are drawn onto the background described by the render options. Screen-space labels (space = "screen") do not use a camera at all, so this is the quick way to turn a label into an image or to decorate an empty canvas; world-space labels need a camera.

Usage

```
render_text(positions, text, camera = NULL, options = NULL, ...)
```

Arguments

positions	Anchor positions, see text_layer .
text	Character vector of labels, see text_layer .
camera	A camera list from camera() or camera_auto() . Ignored for screen-space labels; when NULL, a default camera is used.
options	Render options from render_options() .
...	Further arguments passed to text_layer (for example size, space, color, halo_color).

Value

A list with components width, height, and pixels (raw vector of RGBA values).

See Also

[text_layer](#), [render_scene](#)

Examples

```
img <- render_text(c(20, 20), "figure A", space = "screen",
                  adj = c(0, 1), size = 24,
                  options = render_options(width = 300, height = 80))
tmp <- tempfile(fileext = ".png")
write_png(img, tmp)
```

render_triangles	<i>Render raw triangles without index buffer</i>
------------------	--

Description

Renders triangle geometry where positions and colours are given as flat arrays with 3 vertices per triangle (no index buffer). Useful for voxel renderings, misc3d isosurfaces, and other dynamically generated geometry.

Usage

```
render_triangles(positions, colors, camera, options = render_options())
```

Arguments

positions	Nx3 numeric matrix of vertex positions, where N is a multiple of 3 (3 per triangle).
colors	Nx4 numeric matrix of RGBA colours (0-1 scale).
camera	A camera list from camera() or camera_auto().
options	Render options from render_options().

Value

An image list with width, height, pixels.

Examples

```
# Render a single red triangle from raw vertices
positions <- matrix(c(0, 0, 1, 0, 0, 0.5, 1, 0), ncol = 3, byrow = TRUE)
colors <- matrix(c(1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1), ncol = 4, byrow = TRUE)
cam <- camera_auto(positions)
img <- render_triangles(positions, colors, cam)
```

resample_path	<i>Resample a path at a fixed arc length step</i>
---------------	---

Description

Walks along a path and emits a point every step units of arc length. The result has (nearly) uniform point spacing regardless of how the input was parameterized, which is what makes a swept tube look even: [generate_tube](#) places exactly one cross-section per path point, so uneven spacing means a tube that is finely subdivided in one place and faceted in another. Typical use is to even out the output of [spline_path](#), whose samples are evenly spaced in the curve parameter rather than along the curve.

Usage

```
resample_path(path, step, closed = FALSE)
```

Arguments

path	Nx3 numeric matrix of path points (or a length-3 vector).
step	Arc length between samples. Must be a single positive number; other values return the path unchanged.
closed	Whether the path loops back to its first point (default FALSE).

Details

The spacing that is uniform is the **arc length** along the path; the straight-line distances between consecutive samples are slightly shorter wherever the path curves. An open path always keeps its final point (with a possibly shorter last step); a closed path ends on a copy of its first point and is covered in a whole number of steps, so its effective step can differ from the requested one by a fraction of a percent.

Value

An Nx3 numeric matrix of resampled path points. A path without length (or with fewer than two points) is returned as it is.

See Also

[path_length](#), [spline_path](#)

Examples

```
waypoints <- matrix(c(0, 0, 0, 1, 1, 0, 2, 0, 0), ncol = 3, byrow = TRUE)
smooth <- spline_path(waypoints, samples_per_segment = 32)
even <- resample_path(smooth, step = 0.1)
nrow(even)
```

rotate_mesh

Rotate a mesh around an axis

Description

Vertex normals (if the mesh has any) are rotated with the mesh.

Usage

```
rotate_mesh(mesh, angle_rad, axis = c(0, 0, 1))
```

Arguments

mesh	A mesh descriptor list.
angle_rad	Rotation angle in radians.
axis	Length-3 numeric vector defining the rotation axis.

Value

A new mesh descriptor list with rotated vertices.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
rotated <- rotate_mesh(mesh, pi / 4, axis = c(0, 1, 0))
rotated$vertices[1, ]
```

scale_mesh	<i>Scale a mesh uniformly or per-axis</i>
------------	---

Description

Per-vertex normals (if the mesh has any) are scaled as well, using the inverse transpose of the scaling matrix: a non-uniform scale would otherwise leave normals pointing in a direction that no longer matches the surface, which shows up as wrong shading.

Usage

```
scale_mesh(mesh, scale)
```

Arguments

mesh	A mesh descriptor list.
scale	A single numeric scale factor (uniform) or a length-3 numeric vector for per-axis scaling (x, y, z).

Value

A new mesh descriptor list with scaled vertices.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
big <- scale_mesh(mesh, 3)
flat <- scale_mesh(mesh, c(2, 0.5, 1))
```

 scene

Create a scene descriptor

Description

Bundles a list of meshes together with their placement transforms, a camera, and render options into a single scene object that can be passed to [render_scene\(\)](#) or [write_gltf\(\)](#).

Usage

```
scene(
    meshes,
    camera = NULL,
    options = NULL,
    transforms = NULL,
    names = NULL,
    lines = NULL,
    texts = NULL
)
```

Arguments

meshes	A list of mesh descriptors (scimesh or rgl format, see render_scene()), or a list of already-built scene nodes (<code>list(mesh = ..., transform = ..., name = ...)</code>).
camera	A camera list from camera() or camera_auto() . Optional here; when NULL it must be supplied when calling render_scene() .
options	A render options list from render_options() . Optional here; when NULL, render_scene() uses its default options.
transforms	Optional list of 4x4 numeric matrices, one per mesh, overriding any embedded transform in the nodes. May contain NULL entries to mean identity.
names	Optional character vector, one per mesh, overriding any embedded name.
lines	NULL, a line layer (see line_layer), or a list of them. Line layers are drawn after the meshes with a width measured in pixels and without creating any geometry, which makes them the cheap way to draw many thin lines (wireframes, graph or connectome edges). They share the depth buffer with the meshes and contribute to the scene bounding box (and thus to the camera framing), unless the layer sets <code>affects_bounds = FALSE</code> , see line_layer .
texts	NULL, a text layer (see text_layer), or a list of them. Text layers are drawn after the meshes and the lines as billboards and create no geometry either, so they are the way to annotate a figure (region names, atom labels, panel tags). Unlike line layers they are ignored by the scene bounding box, since their extent depends on the font and the output size.

Details

Each mesh is wrapped into a scene node `list(mesh = ..., transform = ..., name = ...)`. The optional transform is a 4x4 numeric matrix (column-major, GLM style — the same convention accepted by `transform_mesh()`) that places the mesh in world space at render/export time without modifying the mesh itself. The optional name is used by exporters such as glTF for node names.

Value

A scene descriptor list with S3 class "scimesh_scene", with components meshes (list of scene nodes), lines (list of line layers, possibly empty), texts (list of text layers, possibly empty), camera, and options.

See Also

[line_layer](#), [text_layer](#)

Examples

```
cube1 <- generate_cuboid(c(0, 0, 0), c(0.5, 0.5, 0.5), c(1, 0, 0, 1))
cube2 <- generate_cuboid(c(0, 0, 0), c(0.5, 0.5, 0.5), c(0, 0, 1, 1))
# place the second cube 2 units along +X
tr <- diag(1, 4); tr[1, 4] <- 2
sc <- scene(list(cube1, list(mesh = cube2, transform = tr, name = "blue")),
            camera = camera_auto(list(cube1, cube2), direction = c(1, 1, 1)))
img <- render_scene(sc)

# add a line layer drawn on top of the meshes:
sc2 <- scene(list(cube1),
              lines = line_layer(matrix(c(0, 0, 1), ncol = 3),
                                matrix(c(1, 1, 1), ncol = 3), width = 3))

# add a label anchored above the cube:
sc3 <- scene(list(cube1),
              texts = text_layer(matrix(c(0, 1, 0), ncol = 3), "cube",
                                size = 18, adj = c(0.5, 0)))
```

scene_set_line_affects_bounds

Change whether line layers of a scene contribute to its bounds

Description

Sets the `affects_bounds` flag of one or more line layers of a scene, see [line_layer](#). A layer with the flag set contributes to the bounding box of the scene, and thus to the extent that a camera fitted to the scene ([camera_auto](#)) has to cover; a layer without it is ignored when the bounds are computed, which is what you want for decorative lines.

Usage

```
scene_set_line_affects_bounds(
  scene,
  index = NULL,
  name = NULL,
  affects_bounds = TRUE
)
```

Arguments

scene	A scene descriptor list, see scene() .
index	Integer vector, the positions (1-based) of the layers to update, or NULL to select them by name.
name	Character vector, the names of the layers to update, or NULL to select them by index. Bare line layers (which are not wrapped into a scene node) have no name.
affects_bounds	Whether the selected layers contribute to the scene bounds (default TRUE), see line_layer .

Details

The layers can be selected by position (index) or by name (name, for layers that were added as scene nodes with a name). Exactly one of the two has to be given. A scene that contains no mesh at all is framed by its line layers even when they all opted out, since there would otherwise be no geometry to derive a camera from.

Value

The scene with the updated layers, invisibly. Since a scene is a plain list, the update has to be assigned to take effect, e.g. `sc <- scene_set_line_affects_bounds(sc, name = "leader", affects_bounds = FALSE)`.

See Also

[line_layer](#), [camera_auto](#)

Examples

```
from <- matrix(c(-1, 0, 0, 5, 0, 0), ncol = 3, byrow = TRUE)
to <- matrix(c(1, 0, 0, 6, 0, 0), ncol = 3, byrow = TRUE)
# The second segment is a decorative leader line pointing away from the data.
sc <- scene(list(generate_cuboid(c(0, 0, 0), c(0.5, 0.5, 0.5))),
  lines = list(list(lines = line_layer(from, to), name = "edges")))
sc <- scene_set_line_affects_bounds(sc, index = 1, affects_bounds = TRUE)
```

set_mesh_alpha	<i>Set the transparency of a whole mesh</i>
----------------	---

Description

Returns a copy of the mesh in which every vertex (and every face, if per-face colors are used) has the given alpha value. The renderer blends meshes whose colors are not fully opaque automatically, so this is all that is needed to draw a mesh translucently - for example a brain surface at 10 percent opacity for spatial reference.

Usage

```
set_mesh_alpha(mesh, alpha)
```

Arguments

mesh	A mesh descriptor list (see <code>as_scimesh_mesh</code>), e.g. as returned by <code>generate_sphere()</code> or <code>read_ply()</code> .
alpha	Alpha value in $[0, 1]$: 0 = fully transparent, 1 = fully opaque.

Details

A mesh without colors gets uniform colors first (its `default_color` if it has one, light gray otherwise), so the mesh keeps its appearance and only becomes see-through. Use `alpha = 0` for completely invisible geometry and `alpha = 1` to make a mesh opaque again.

Value

A mesh descriptor list with the alpha applied.

See Also

[render_mesh, scene](#)

Examples

```
sphere <- generate_sphere(c(0, 0, 0), 1)
ghost  <- set_mesh_alpha(sphere, 0.2)

# Per-vertex alpha (here: every other vertex transparent) can be set
# directly on the color matrix:
cols <- sphere$colors
cols[, 4] <- rep(c(0, 1), length.out = nrow(cols))
sphere$colors <- cols
```

spline_path

Smooth curve through ordered 3D points

Description

Turns a coarse list of waypoints into a dense, smooth path, which is what the path-taking geometry functions expect: `generate_tube` sweeps a cross-section along a path, and `line_layer` draws one straight segment per consecutive pair of points. Handing a handful of control points to those gives a visibly faceted tube and a polygonal line; this is how you get the smooth version.

Usage

```
spline_path(
    points,
    method = c("catmull-rom", "bspline"),
    samples_per_segment = 8L,
    closed = FALSE,
    alpha = 0.5
)
```

Arguments

<code>points</code>	Nx3 numeric matrix of points the curve has to pass through (or a length-3 vector for a single point). An open curve needs at least 2 points, a closed one at least 3 ("bspline": at least 4).
<code>method</code>	Either "catmull-rom" (interpolating, default) or "bspline" (approximating, smoother; see the description).
<code>samples_per_segment</code>	Number of points to generate per segment between two input points (default 8, clamped to at least 1). This is the knob that controls how smooth the result looks when rendered.
<code>closed</code>	Whether the curve loops back to its first point (default FALSE). A closed path ends on a copy of its first point.
<code>alpha</code>	Parameterization exponent of the Catmull-Rom curve (default 0.5, see the description). Ignored for "bspline".

Details

Two curves are available, and they differ in a way that matters:

- "catmull-rom" (the default) **interpolates**: the curve passes through every input point, and each point's tangent is derived from its two neighbours. This is what you want when the input points are positions that the curve has to hit (measured or digitized data).
- "bspline" **approximates**: the points act as a control cage that the curve stays inside, which irons out noise instead of reproducing it, and the curve is C^2 continuous everywhere. It generally does not pass through the input points.

For the Catmull-Rom curve, alpha selects the parameterization and is the knob that matters most on real data:

- 0.5 (default) — **centripetal**. Knot intervals grow with the square root of the chord length, which prevents the cusps, loops and self-intersections that the uniform curve produces when the spacing of the input points is uneven (long segment followed by a short one, the norm for measured data).
- 1 — chord length: still free of cusps, but can overshoot more.
- 0 — uniform: the textbook curve, exact on evenly spaced points and misbehaving on everything else.

Value

An Nx3 numeric matrix of path points, ready for [generate_tube](#), [generate_tubes](#), [line_layer](#) or [camera_auto](#). An empty (0-row) matrix if there are not enough distinct points for the chosen curve.

See Also

[bezier_path](#), [resample_path](#), [path_curvature](#), [generate_tube](#)

Examples

```
waypoints <- matrix(c(0, 0, 0, 1, 1, 0, 2, 0, 0, 3, 1, 0),
                    ncol = 3, byrow = TRUE)
path <- spline_path(waypoints, samples_per_segment = 8)
nrow(path) # 3 segments * 8 samples + the final point

# A smooth tube through the waypoints, instead of a faceted one:
mesh <- generate_tube(waypoints, radius = 0.1)
smooth <- generate_tube(path, radius = 0.1)

# An approximating (noise-reducing) closed loop:
loop <- spline_path(waypoints, method = "bspline", closed = TRUE)
```

stack_horizontal

Stack images horizontally

Description

Stacks a list of rendered images side by side. A convenience wrapper around `compose_layout()`.

Usage

```
stack_horizontal(
  ...,
  colorbar = NULL,
  colorbar_height = 80L,
  colorbar_width = 80L,
  background = c(0, 0, 0, 0),
  colorbar_side = c("right", "left"),
  crop = FALSE
)
```

Arguments

<code>...</code>	Images from <code>render_mesh()</code> or <code>render_scene()</code> , or a list of images.
<code>colorbar</code>	Optional colorbar.
<code>colorbar_height</code>	Height of the colorbar in pixels.
<code>colorbar_width</code>	Width of the colorbar in pixels.
<code>background</code>	Background RGBA color for padding.
<code>colorbar_side</code>	Side for the colorbar: "right" (default) or "left".
<code>crop</code>	If TRUE, crop whitespace from the output.

Value

A composed image list.

Examples

```
mesh1 <- generate_cuboid(c(-2, 0, 0), c(0.5, 1, 1), c(1, 0, 0, 1))
mesh2 <- generate_cuboid(c( 2, 0, 0), c(0.5, 1, 1), c(0, 0, 1, 1))
img1 <- render_mesh(mesh1$vertices, mesh1$triangles)
img2 <- render_mesh(mesh2$vertices, mesh2$triangles)
result <- stack_horizontal(img1, img2)
tmp_file <- tempfile(fileext = ".png")
write_png(result, tmp_file)
```

<code>stack_vertical</code>	<i>Stack images vertically</i>
-----------------------------	--------------------------------

Description

Stacks a list of rendered images vertically (one below another). A convenience wrapper around `compose_layout()`.

Usage

```
stack_vertical(  
  ...,  
  colorbar = NULL,  
  colorbar_height = 80L,  
  colorbar_width = 80L,  
  background = c(0, 0, 0, 0),  
  colorbar_side = c("right", "left"),  
  crop = FALSE  
)
```

Arguments

- ... Images from render_mesh() or render_scene(), or a list of images.
- colorbar Optional colorbar from colorbar_horizontal() or colorbar_vertical().
- colorbar_height Height of the colorbar in pixels.
- colorbar_width Width of the colorbar in pixels.
- background Background RGBA color for padding.
- colorbar_side Side for the colorbar: "right" (default) or "left".
- crop If TRUE, crop whitespace from the output.

Value

A composed image list.

Examples

```
mesh1 <- generate_cuboid(c(0, 2, 0), c(1, 0.5, 1), c(1, 0, 0, 1))  
mesh2 <- generate_cuboid(c(0, -2, 0), c(1, 0.5, 1), c(0, 1, 0, 1))  
img1 <- render_mesh(mesh1$vertices, mesh1$triangles)  
img2 <- render_mesh(mesh2$vertices, mesh2$triangles)  
result <- stack_vertical(img1, img2)  
tmp_file <- tempfile(fileext = ".png")  
write_png(result, tmp_file)
```

text_extent	Measure text
-------------	--------------

Description

Computes the size of the text box that `text_layer` uses, which is needed to place labels relative to each other (for example to right-align a caption, or to keep two labels from overlapping). Multi-line labels are measured as a whole, using the same line spacing as the renderer.

Usage

```
text_extent(text, size = 18, font_file = NULL, line_spacing = 1.2)
```

Arguments

`text` Character vector of labels (UTF-8; "\n" for line breaks).
`size` Text height in output pixels (default 18).
`font_file` Path to a .ttf file, or NULL for the bundled font (see [default_font\(\)](#)).
`line_spacing` Distance between lines, as a multiple of the glyph box height (default 1.2).

Value

A data frame with one row per input string and the columns `text`, `width` (widest line), `height` (whole block), `ascent`, `descent` and `lines`.

See Also

[text_layer](#)

Examples

```
text_extent("anterior", size = 20)
text_extent(c("left hemisphere", "right hemisphere"), size = 16)
```

text_layer	<i>Create a text label layer (2D annotations for a scene)</i>
------------	---

Description

Bundles strings with anchor positions into a layer that can be added to a scene (see the `texts` argument of [scene\(\)](#)), or drawn directly with [render_text](#).

Usage

```
text_layer(
  positions,
  text,
  colors = NULL,
  size = 18,
  font_file = NULL,
  space = c("world", "screen"),
  adj = c(0.5, 0.5),
  offset = c(0, 0),
  line_spacing = 1.2,
  depth_test = TRUE,
  halo_color = NULL,
```

```

    halo_width = 1.5,
    rotation = 0
)

```

Arguments

positions	Anchor positions: an Nx3 numeric matrix for world space, or an Nx2/Nx3 numeric matrix for space = "screen" (third column ignored). A single point may be given as a numeric vector of length 2 or 3.
text	Character vector of labels (UTF-8), recycled to nrow(positions). Use "\n" for line breaks.
colors	RGBA colour(s): a single vector applied to all labels, or an Nx4 numeric matrix (values in [0, 1], alpha optional). The default NULL uses the default_color of the render options.
size	Text height in output pixels (default 18). Independent of the anti-aliasing setting: a label keeps its physical size when aa_samples is raised.
font_file	Path to a .ttf file to use. NULL (the default) uses the bundled Inter font, see default_font() .
space	"world" (default) for positions in the 3D scene, or "screen" for positions in output pixels.
adj	Numeric vector of length 2 giving where the position sits on the text box, in [0, 1]: c(0, 0) is the bottom left corner, c(1, 1) the top right one, and the default c(0.5, 0.5) centres the text on the position.
offset	Numeric vector of length 2: extra offset in output pixels (positive x = right, positive y = down), applied after anchoring. Handy to push an atom label next to the atom instead of onto it.
line_spacing	Distance between two lines of a multi-line label, as a multiple of the font's glyph box height (default 1.2).
depth_test	Whether a label is hidden by geometry in front of its anchor (default TRUE). Set to FALSE to always draw the labels on top of everything, which is the right choice for direction annotations ("anterior") and for labels on a surface.
halo_color	RGBA colour of the halo (outline) drawn behind the glyphs, which keeps labels readable on dark or busy geometry. NULL (the default) draws no halo.
halo_width	Halo thickness in pixels (default 1.5).
rotation	Rotation of the label in degrees, counter-clockwise, about the anchor position (default 0). Use 90 to write along a vertical axis (the usual orientation of a y-axis label), 180 for an upside-down label, or any other angle to follow an annotation line. The anchor stays fixed while the text turns around it.

Details

Labels are *billboards*: they always face the camera and keep the size given in size (in output pixels), so they stay readable from any viewpoint — unlike text that is turned into 3D geometry, which skews as the camera moves. This is the screen-friendly counterpart of `rgl::text3d()`, and the intended way to label brain regions, atoms, panels or figure axes.

Positions are given either in **world space** (the default) or in **screen space**: with `space = "screen"` the coordinates are pixels of the output image, measured from the top left corner, which is what you want for titles, panel tags and captions. World-space labels are projected with the camera of the scene, so they stick to the annotated location, and they are hidden by geometry in front of them unless `depth_test = FALSE`.

Value

A text layer object (a list with class `scimesh_text`) for use in `scene()` or `render_text`.

See Also

[render_text](#), [text_extent](#), [scene](#), [line_layer](#)

Examples

```
sph <- generate_sphere(c(0, 0, 0), radius = 1)
# world-space label above the sphere
labels <- text_layer(matrix(c(0, 1.4, 0), ncol = 3), "top", size = 20)
labels

# screen-space panel tag, positioned by its top left corner
tag <- text_layer(c(10, 12), "A", space = "screen", adj = c(0, 1), size = 28,
                  halo_color = c(1, 1, 1, 0.9))
tag

# multiple labels with per-label colors and positions
multi <- text_layer(matrix(c(-1, 0, 0, 1, 0, 0), ncol = 3, byrow = TRUE),
                    c("left", "right"), colors = matrix(c(1, 0, 0, 1,
                                                            0, 0, 1, 1),
                                                            ncol = 4, byrow = TRUE))
multi
```

transform_mesh

Apply a 4x4 transformation matrix to a mesh

Description

Transforms all vertex positions in a mesh by a 4x4 homogeneous matrix (applied as $M * (x, y, z, 1)^T$). Vertex colors are kept as they are; vertex normals (if the mesh has any) are transformed by the inverse transpose of M , so that shading stays correct for shearing and non-uniform scaling. Use `compute_vertex_normals()` if the mesh has no normals yet.

Usage

```
transform_mesh(mesh, matrix)
```

Arguments

- mesh** A mesh descriptor list with vertices and triangles, as returned by `render_mesh()` or built by `scimesh_generate_multi_spheres()` etc.
- matrix** A 4x4 numeric matrix.

Value

A new mesh descriptor list with transformed vertices.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
mat <- diag(4)
mat[1:3, 4] <- c(2, 3, 4)
translated <- transform_mesh(mesh, mat)
translated$vertices[1, ]
```

translate_mesh	<i>Translate a mesh</i>
----------------	-------------------------

Description

Vertex colors and normals are untouched: a translation does not change the orientation of a surface.

Usage

```
translate_mesh(mesh, translation)
```

Arguments

- mesh** A mesh descriptor list.
- translation** Length-3 numeric vector (x, y, z).

Value

A new mesh descriptor list with translated vertices.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
moved <- translate_mesh(mesh, c(5, 0, 0))
colMeans(moved$vertices)
```

viridis_colormap	<i>Viridis colormap</i>
------------------	-------------------------

Description

Returns the viridis color palette. A convenience wrapper around `grDevices::hcl.colors` that mimics the viridis color scheme without requiring extra packages.

Usage

```
viridis_colormap(n, alpha = 1, direction = 1)
```

Arguments

<code>n</code>	Number of colors.
<code>alpha</code>	Alpha channel value (0-1).
<code>direction</code>	Forward (1) or reversed (-1) direction.

Value

A character vector of hex color strings.

Examples

```
cols <- viridis_colormap(10)
plot(1:10, pch = 19, col = cols, cex = 3)
```

world_to_screen	<i>Project world coordinates to image pixels</i>
-----------------	--

Description

Runs the same view and projection as the renderer, so the result lands exactly on the rendered image. Useful to place screen-space annotations (`text_layer(space = "screen")`) next to a 3D location, or to draw callout lines with `line_layer` in image coordinates.

Usage

```
world_to_screen(points, camera, width, height, options = NULL)
```

Arguments

points	An Nx3 numeric matrix of world coordinates (or a length-3 vector for a single point).
camera	A camera list from camera() or camera_auto() .
width, height	Size of the rendered image in pixels.
options	Render options, used for the projection type and the clipping planes. NULL uses default options with the given size.

Value

A data frame with one row per input point and the columns x, y (pixels, origin top left), depth (NDC depth, smaller is closer) and in_front (whether the point is in front of the camera; for FALSE the pixel coordinates are not meaningful).

See Also

[text_layer](#)

Examples

```
sph <- generate_sphere(c(0, 0, 0), radius = 1)
cam <- camera_auto(list(sph), direction = c(0, 0, 1))
opts <- render_options(width = 400, height = 300)
world_to_screen(matrix(c(0, 1, 0), ncol = 3), cam, 400, 300, opts)
```

write_gltf

Write a scene or mesh list to a glTF file

Description

Exports a scene (or a list of meshes) to the glTF 2.0 format, either as a JSON document with an external binary buffer (.gltf + .bin) or as a single self-contained binary file (.glb). The resulting file can be viewed in any glTF-capable viewer (e.g. a browser using three.js) and includes per-mesh placement transforms, vertex colors, and optionally a camera.

Usage

```
write_gltf(meshes, path, camera = NULL, format = c("gltf", "glb"))
```

Arguments

meshes	Either a scimesh_scene object (see scene()), a list of mesh descriptors, or a list of scene nodes — the same inputs accepted by render_scene() .
path	Output file path. For format = "gltf" the binary buffer is written next to it as <stem>.bin.

camera	Optional camera list from <code>camera()</code> or <code>camera_auto()</code> . When provided, a glTF perspective camera node is included.
format	Output format: "gltf" (default, JSON + .bin) or "glb" (single binary file).

Details

Renderer-specific settings (shading mode, fog, SSAO, ...) are not part of the glTF standard and are not exported. Per-face colors are exported by splitting vertices (each triangle gets its own vertices), which increases geometry roughly 3x.

Value

Invisibly NULL.

Examples

```
cube <- generate_cuboid(c(0, 0, 0), c(0.5, 0.5, 0.5), c(1, 0, 0, 1))
tr <- diag(1, 4); tr[1, 4] <- 2
sc <- scene(list(cube, list(mesh = cube, transform = tr, name = "second")))
out <- tempfile(fileext = ".glb")
write_gltf(sc, out, format = "glb")
```

write_png	<i>Write a rendered image to a PNG file</i>
-----------	---

Description

Writes the output of `render_mesh()` or `render_scene()` to a PNG file using the built-in C++ PNG writer (`stb_image_write`). No additional R packages are required.

Usage

```
write_png(image, filename)
```

Arguments

image	An image list returned by <code>render_mesh()</code> or <code>render_scene()</code> .
filename	Output PNG file path.

Value

No return value; called for side effects.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(mesh$vertices, mesh$triangles)
tmp_file <- tempfile(fileext = ".png")
write_png(img, tmp_file)
```

write_stl	<i>Write a mesh to an STL file</i>
-----------	------------------------------------

Description

Writes a scimesh mesh descriptor to an ASCII or binary STL file.

Usage

```
write_stl(mesh, path, format = c("binary", "ascii"))
```

Arguments

mesh	A mesh descriptor list.
path	Path to the output STL file.
format	"binary" (default) or "ascii".

Value

invisible NULL, called for side effects of writing the file.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
tmp_file <- tempfile(fileext = ".stl")
write_stl(mesh, tmp_file, format = "binary")
```

`write_tga`*Write a rendered image to a TGA file*

Description

Writes the output of `render_mesh()` or `render_scene()` to a TGA file using scimesh's own C++ TGA writer (no external dependencies). TGA output is uncompressed true-color.

Usage

```
write_tga(image, filename, use24bit = FALSE)
```

Arguments

<code>image</code>	An image list returned by <code>render_mesh()</code> or <code>render_scene()</code> .
<code>filename</code>	Output TGA file path.
<code>use24bit</code>	If TRUE, write 24-bit RGB (no alpha channel). The default FALSE writes 32-bit RGBA.

Value

No return value; called for side effects.

Examples

```
mesh <- generate_cuboid(c(0, 0, 0), c(1, 1, 1))
img <- render_mesh(mesh$vertices, mesh$triangles)
tmp_file <- tempfile(fileext = ".tga")
write_tga(img, tmp_file)
```

Index

`apply_colormap`, 4
`as_scimesh_mesh`, 17

`bezier_path`, 5, 63

`camera`, 6, 9, 52, 54, 58, 71, 72
`camera_auto`, 7, 8, 9, 52, 54, 58–60, 63, 71, 72
`camera_fit_scene`, 8, 8
`camera_orbit`, 9
`clip_plane`, 10, 48
`colorbar_horizontal`, 11
`colorbar_vertical`, 13
`compose_layout`, 14
`compute_vertex_normals`, 15, 42

`default_font`, 16, 18, 66, 67
`diverging_colormap`, 16

`flip_uv`s, 17, 45
`font_info`, 16, 18

`generate_arrow`, 19
`generate_axes`, 20
`generate_bbox`, 20
`generate_cone`, 21
`generate_cuboid`, 22
`generate_cylinder`, 22, 24, 29
`generate_multi_cylinders`, 23, 25, 30
`generate_multi_spheres`, 24
`generate_plane`, 25
`generate_pyramid`, 26
`generate_sphere`, 25, 27
`generate_tetrahedron`, 27
`generate_torus`, 28
`generate_tube`, 5, 29, 30, 41, 55, 62, 63
`generate_tubes`, 24, 29, 30, 36, 38, 63

`image_apply_contrast`, 31
`image_crop`, 32
`image_crop_to_content`, 32
`image_grow`, 33

`image_merge`, 34
`image_rotate_90`, 35
`image_scale`, 35
`image_to_array`, 36

`line_layer`, 8, 36, 51, 52, 58–60, 62, 63, 68, 70

`mesh_bbox`, 38
`mesh_from_rgl`, 39, 45
`mesh_to_rgl`, 39

`path_curvature`, 40, 63
`path_length`, 41, 56

`read_obj`, 42
`read_ply`, 42
`read_stl`, 43
`render_lines`, 44
`render_mesh`, 17, 42, 45, 61
`render_options`, 11, 46, 52, 54, 58
`render_points`, 49, 52
`render_scene`, 50, 52, 54, 58, 71
`render_segments`, 36–38, 51
`render_spheres`, 53
`render_text`, 54, 66, 68
`render_triangles`, 55
`resample_path`, 41, 55, 63
`rotate_mesh`, 56

`scale_mesh`, 57
`scene`, 8, 9, 36, 37, 51, 58, 60, 61, 66, 68, 71
`scene_set_line_affects_bounds`, 9, 37, 59
`set_mesh_alpha`, 46, 61
`spline_path`, 5, 41, 55, 56, 62
`stack_horizontal`, 63
`stack_vertical`, 64

`text_extent`, 16, 65, 68
`text_layer`, 16, 18, 51, 54, 58, 59, 65, 66, 66, 70, 71

`transform_mesh`, [59](#), [68](#)

`translate_mesh`, [69](#)

`viridis_colormap`, [70](#)

`world_to_screen`, [70](#)

`write_gltf`, [58](#), [71](#)

`write_png`, [72](#)

`write_stl`, [73](#)

`write_tga`, [74](#)