Maquette

Render 3D models in Typst

github.com/bernsteining/maquette · typst.app/universe/package/maquette · bernsteining.github.io/maquette

Version 0.1.2 · July 26, 2026

CONTENTS

Contents

Introduction

Maquette is a Typst plugin for rendering 3D models directly inside your documents. It loads STL, OBJ, and PLY files and produces publication-ready images — no external renderer, no screenshots, no manual exporting. Everything runs with WASM.

Under the hood, Maquette is a small rasterizer with a real lighting pipeline: multi-light Blinn-Phong shading, Fresnel reflections, subsurface scattering, ambient occlusion (SSAO), bloom, tone mapping, and more. Models can be rendered to PNG (rasterized, constant-size output) or SVG (scalable vector polygons). The full configuration — camera, lights, materials, post-processing — lives in your `.typ` source, so every view is reproducible and version-controllable.

Quickstart

Import a render function, read a model file, and call it. That's it.

```
#import "@preview/maquette:0.1.2": render-stl

#let cube = read("data/cube.stl", encoding: none)
#render-stl(cube)
```

For STL & PLY: Always read with `encoding: none`. Without it, Typst's `read()` defaults to UTF-8 text and *binary STL/PLY files* — or any file containing non-UTF-8 bytes — fail with a *"file is not valid UTF-8"* error before maquette even runs. For OBJ, it shouldn't be necessary.

The default output is PNG; pass `format: "svg"` for vector output:

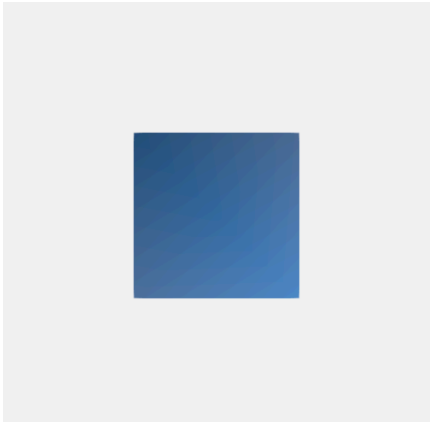
```
#render-stl(cube, format: "svg")
```

Inline Show Rule

With a show rule, you can write OBJ / STL / PLY geometry directly in fenced code blocks and have it rendered inline:

```
#show raw.where(lang: "obj"): it => {
  render-obj(bytes(it.text))
}
```

````
```obj
v  0.0  1.0  0.0
v -0.5  0.0 -0.5
v  0.5  0.0 -0.5
v  0.5  0.0  0.5
v -0.5  0.0  0.5
f 1 3 2
f 1 4 3
f 1 5 4
f 1 2 5
f 2 3 4 5
```
````



Config Reference

All parameters are optional — pass them as named arguments or a dictionary; defaults are shown below. Setting any to `none` restores its default (for `background`, that means transparent).

```
{ // ── Camera & Viewport ──────────────────────────────────────
  "camera": [3, 3, 3],                          // Camera position in world space (Cartesian)
  "azimuth": null,                              // Spherical camera: horizontal angle in degrees
  "elevation": null,                            // Spherical camera: vertical angle in degrees
  "distance": null,                             // Spherical camera: distance from center (auto)
  "center": [0, 0, 0],                          // Look-at target (overridden by auto_center)
  "up": [0, 0, 1],                              // Up direction vector
  "fov": 45,                                    // Vertical FOV in degrees (perspective only)
  "projection": "perspective",                  // "perspective", "orthographic", "isometric" ...
  "auto_center": true,                          // Auto-center on model bounding box
  "auto_fit": true,                             // Scale model to fill viewport
  "zoom": 1.0,                                  // Multiplier on auto-fit scale (>1 zooms in)
  "pan": [0, 0],                                // Screen-space recentring [right, up] as viewport fraction
  "width": 500,                                 // Output width in pixels
  "height": 500,                                // Output height in pixels
  "background": "#f0f0f0",                      // Background color (hex); none, "none" or "" = transparent
  // ── Appearance ──────────────────────────────────────────────
  "color": "#4488cc",                           // Model fill color (hex)
  "stroke": {"color": "none", "width": 0},      // Triangle edge stroke (or just "#hex")
  "light_dir": [1, 2, 3],                       // Directional light vector
  "ambient": 0.15,                              // Ambient light intensity (0-1)
  "mode": "solid",                              // "solid", "wireframe", "solid+wireframe", "x-ray"
  "xray_opacity": 0.1,                          // Front-face opacity for x-ray mode (0-1)
  "cull_backface": true,                        // Back-face culling (auto-disabled for x-ray)
  "wireframe": {"color": "", "width": 1.0},     // Wireframe edges (or just "#hex")
  "smooth": true,                               // Gouraud smooth shading (best with PNG)
  "specular": 0.2,                              // Specular highlight intensity (0-1)
  "shininess": 32,                              // Specular exponent (higher = tighter)
  "gamma_correction": true,                     // Compute lighting in linear sRGB space
  "fresnel": {"intensity": 0.3, "power": 5},    // Fresnel rim lighting (or just 0.3)
  "sss": false,                                 // true or {intensity, power, distortion}
  "opacity": 1.0,                               // Global opacity (0-1)
  "lights": [],                                 // [{type: directional|positional|area, vector, color, ...]
  "tone_mapping": {"method": "", "exposure": 1.0}, // HDR tone mapping (or just "aces")
  "shading": "",                                // "blinn-phong" (default), "gooch", "cel", "flat", "normal"
  "gooch_warm": "#ffcc44",                      // Gooch warm tone color
  "gooch_cool": "#4466cc",                      // Gooch cool tone color
  "cel_bands": 4,                               // Number of cel-shading bands
  "materials": {},                              // OBJ material map: { "name": "#hex" }
  "highlight": {},                              // OBJ group highlight: "#hex" or {color, specular, ...}
  // ── Annotations ─────────────────────────────────────────────
  "annotations": false,                         // true or {groups, color, font_size, offset}
  // ── Color Mapping ───────────────────────────────────────────
  "color_map": "",                              // "overhang", "curvature", "scalar", or "" (off)
  "color_map_palette": [],                      // Custom hex color gradient (curvature/scalar)
  "scalar_function": "",                        // Math expression for scalar mode: "sqrt(x*x+y*y+z*z)"
  "vertex_smoothing": 4,                        // Smooth color values across vertices (0-4)
  "overhang_angle": 45,                         // Overhang threshold in degrees
  // ── Outlines ────────────────────────────────────────────────
  "outline": false,                             // true or {color, width}
  // ── Effects ─────────────────────────────────────────────────
  "ground_shadow": false,                       // true or {opacity, color}
  "shadows": false,                             // Cast/self shadows: true or {per_pixel, softness, color, omni, ...}
  "clip": null,                                 // Cut plane: (a,b,c,d) or {from|axis|normal, depth, keep, cap, hatch}
  "explode": 0,                                 // Exploded view factor
  "decimate": 0,                                // Mesh simplification 0-1 (higher = fewer triangles)
  "point_size": 0,                              // Point cloud neighbor radius (0 = auto)
  "antialias": 1,                               // 0: off, 1: FXAA, 2: SSAA, 3-4: SSAA x2
  "ssao": false,                                // true or {samples, radius, bias, strength}
  "bloom": false,                               // true or {threshold, intensity, radius}
  "glow": false,                                // true or {color, intensity, radius}
  "sharpen": false,                             // true or {strength} (default 0.5)
  // ── Multi-View ──────────────────────────────────────────────
  "views": null,                                // Named views: ["front", "right", "top", ...]
  "turntable": {"iterations": 0, "elevation": 40}, // Turntable views (or just 6)
  "grid_labels": true,                          // Show labels on multi-view grids
  // ── Diagnostics ─────────────────────────────────────────────
  "debug": false,                               // Overlay model metadata
  "debug_color": "#cc2222"                      // Debug text color
}
```

Appearance

Custom Color

Change the global color of the model by changing the `color` field.

```
#render-stl(cube,
  color: "#c0ffee",
  width: 60%,
)
```

Background Color

Set `background` to a hex color to fill the image background.

```
#render-stl(cube,
  center: (0.5, 0.5, 0.5),
  background: "#1a1a2e",
  width: 60%,
)
```

Set `background` to `none` (the string `"none"` and an empty string `""` work too) for a transparent background — the PNG carries a real alpha channel, so the model blends into the page.

```
#render-stl(cube,
  background: none,
  width: 60%,
)
```

The cube looks like a flat square from this angle — let's learn how to change the point of view.

Camera Position

Cartesian coordinates

Change the camera position and where it points to using cartesian coordinates with `camera:(x,y,z)` and `center:(x,y,z)`. As you may have noticed with previous examples, Maquette finds the model's bounding box automatically and points the camera at its centre — that is `auto_center: true`, the default — so most of the time no `center` is needed. Set `auto_center: false` to aim at the explicit `center` you provide instead.

```
#render-obj(teapot,
  camera: (0, 0, 10),
  up: (0, 1, 0),
)
```

The `up` parameter defines which direction points "up" in the scene. The default is `(0, 0, 1)` (Z-up), which matches the convention used by most CAD software and STL files. OBJ files exported from Blender, game engines, or other Y-up tools typically need `up: (0, 1, 0)` to display correctly.

Spherical coordinates

Instead of placing the camera with Cartesian `(x, y, z)` coordinates, you can use `azimuth` (horizontal angle) and `elevation` (vertical angle) in degrees. This makes it much easier to orbit around a model — just change the angles. When either is set, they override the `camera` field. The `distance` is auto-computed from the bounding box unless specified.

```
#render-obj(teapot,
  up: (0, 1, 0),
  azimuth: 30,
  elevation: -10,
  distance: 10,
)
```

🎥 **Placing the camera, fast.** Hunting for the right angles by editing numbers and recompiling is slow. The live browser demo (https://bernsteining.github.io/maquette/) runs the identical WASM but your browser *JIT-compiles* it to native code, so it iterates far faster: drag to orbit, scroll to zoom, then paste the generated snippet straight into your document!

Field of View

```
#render-obj(teapot,
  up: (0, 1, 0),
  fov: 20,
  width: 60%,
)
```

```
#render-obj(teapot,
  up: (0, 1, 0),
  azimuth: 45,
  distance: 30,
  auto_fit: false,
  fov: 10,
  width: 60%,
)
```

The `fov` parameter controls the vertical field of view angle (in degrees) for perspective projection. Lower values produce a telephoto effect, higher values create wide-angle distortion. Default is 45. By default (`auto_fit: true`), Maquette scales the model to fill the viewport; set `auto_fit: false` to use raw world-space coordinates, which lets you control framing manually with `distance` and `fov`.

Framing — Zoom & Pan

`auto_fit` fits the model's bounding **sphere** to the viewport, so broad or spread-out models leave empty margins — this teapot is wide and flat, so fitting its sphere to the frame width strands generous space above and below it. `zoom` multiplies the fit scale to reclaim that space; `pan: (right, up)` then shifts the model in screen space (as a fraction of the viewport), so a tighter zoom can be recentred to taste.

`zoom: 1.0 (default)`

`zoom: 1.45`

`zoom: 1.45, pan: (-0.12, 0)`

At `zoom: 1.45` the teapot fills the frame vertically, but its spout and handle now press against both side edges; `pan: (-0.12, 0)` slides it left, tucking the handle in from the right edge. Both default to no-ops (`zoom: 1.0`, `pan: (0, 0)`), so existing renders are unaffected.

Projections

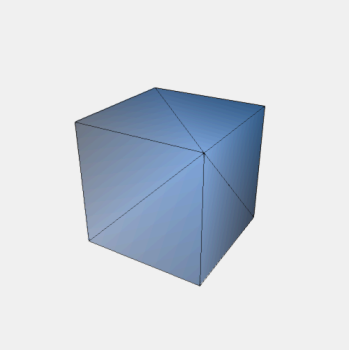
Maquette supports 14 projection types. Set `projection: "name"` to switch.

In the following examples we're using `stroke: (color, width)` to visualize triangle edges, in order to better visualize each projection's property.

Perspective (default)

Objects farther from the camera appear smaller, giving a natural sense of depth.

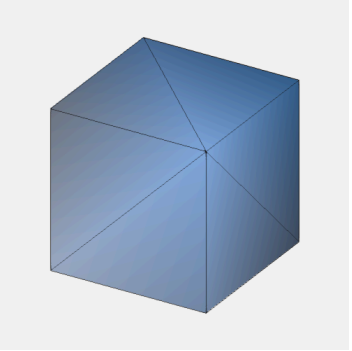
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke: (color: "#111111",
width: 1.0),
)
```



Orthographic

No perspective foreshortening: parallel lines stay parallel regardless of distance.

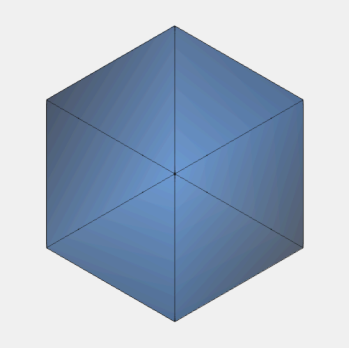
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "orthographic",
)
```



Isometric

Three principal axes appear equally foreshortened. Common in technical illustrations and game art.

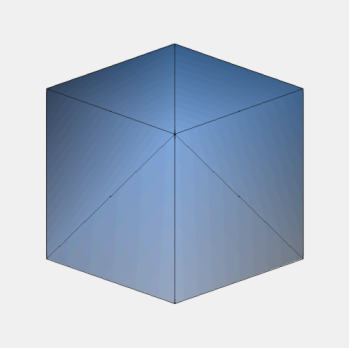
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "isometric",
)
```



Dimetric

Two axes equally foreshortened, the third differs. Elevation 20.7°, azimuth 45°.

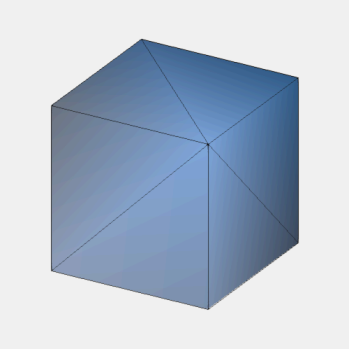
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "dimetric",
)
```



Trimetric

All three axes have different foreshortening. Elevation 25°, azimuth 30°.

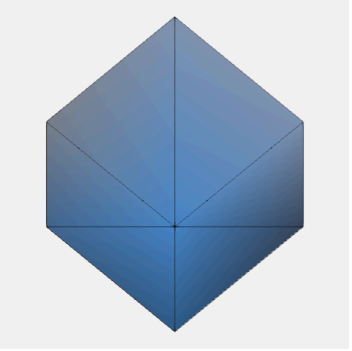
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "trimetric",
)
```



Military

Top-down axonometric where the plan view dominates. Elevation 54.7°, azimuth 45°.

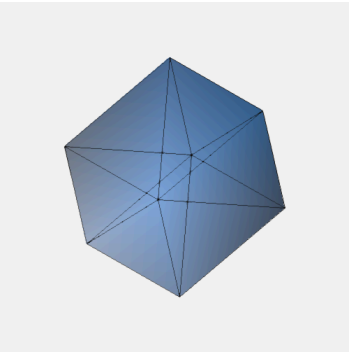
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "military",
)
```



Cabinet

Oblique projection: front face at true shape, depth axis at half scale, 45° angle.

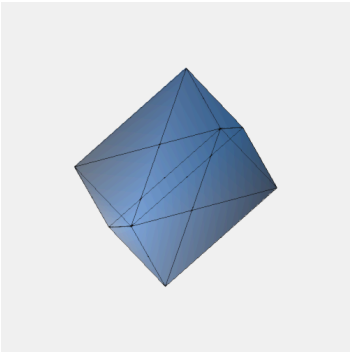
```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "cabinet",
)
```



Cavalier

Like cabinet, but the depth axis is drawn at full scale at a 45° angle. All dimensions are preserved equally.

```
#render-stl(cube,
  camera: (3, 2, 2),
  stroke:
    (color: "#111111",
     width: 1.0),
  projection: "cavalier",
)
```



Fisheye

Equidistant: angular distance maps linearly to radius. Uniform distortion across the field.

```
#render-obj(teapot,
  up: (0, 1, 0),
  elevation: 25,
  distance: 2.5,
  projection: "fisheye",
)
```

Stereographic

Conformal: preserves local shapes but enlarges the periphery. Compare the teapot's spout/handle to fisheye.

```
#render-obj(teapot,
  up: (0, 1, 0),
  elevation: 25,
  distance: 2.5,
  projection:
"stereographic",
)
```

Curvilinear

Perspective with barrel distortion: straight lines curve outward near the edges, simulating a wide-angle lens.

```
#render-obj(teapot,
  up: (0, 1, 0),
  elevation: 25,
  distance: 14,
  projection: "curvilinear",
)
```

Cylindrical

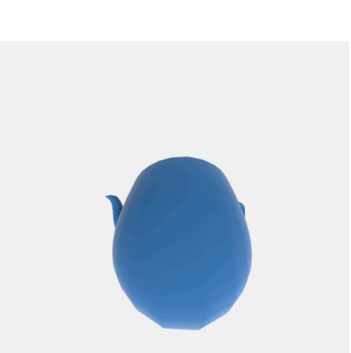
Horizontal angles map linearly (like a panorama), vertical stays perspective. Keeps vertical lines straight.

```
#render-obj(teapot,
  up: (0, 1, 0),
  elevation: 25,
  distance: 2.5,
  projection: "cylindrical",
)
```

Pannini

Architectural photography projection: verticals stay straight, horizontals curve gracefully. A hybrid between cylindrical and stereographic.

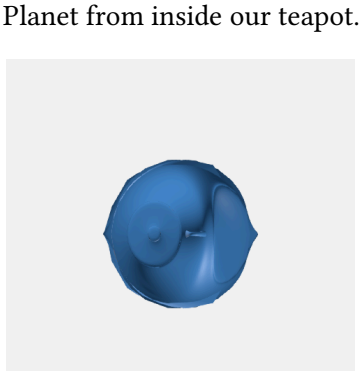
```
#render-obj(teapot,
  up: (0, 1, 0),
  distance: 2.5,
  projection: "pannini",
)
```



Tiny Planet

Full 360° inverse projection: objects ahead wrap to the outer edge, objects behind map to the center. Backface culling auto-disabled. Example shows Tiny Planet from inside our teapot.

```
#render-obj(teapot,
  camera: (0, 1.7, 0),
  up: (0, 0, 1),
  projection: "tiny-planet",
)
```



Shading & Lighting

Ambient & Light Direction

The `ambient` parameter (0–1) controls how much light reaches surfaces regardless of their orientation. Low values create dramatic contrast; high values flatten the shading. The `light_dir` vector sets the direction light comes from.

`ambient: 0.05`

`ambient: 0.3`

`ambient: 0.6`

Hemisphere Ambient

The `ambient` parameter accepts either a number (flat ambient, as before) or an object with `intensity`, `sky`, and `ground` fields. Hemisphere ambient lerps between a sky color (for upward-facing normals) and a ground color (for downward-facing normals), simulating environmental lighting without any extra cost.

Defaults:

```
ambient: (intensity: 0.15, sky: "#ccd4e0", ground: "#d4ccc4")
```

Flat ambient (default)

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  specular: 0.3,
  ambient: 0.3,
  width: 100%,
)
```

Hemisphere ambient

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  specular: 0.3,
  ambient: (intensity: 0.4, sky: "#8899cc", ground: "#443322"),
  width: 100%,
)
```

Smooth Shading

Smooth shading is enabled by default. Vertex normals are averaged across adjacent faces and lighting is interpolated per-pixel (Gouraud shading), smoothing out the faceted appearance. Set `smooth: false` to revert to flat shading, where each triangle gets a single color based on its face normal.

Smooth (default)

Flat (`smooth: false`)

Specular Highlights

Add Blinn-Phong specular highlights with the `specular` parameter (0–1). The `shininess` exponent controls how tight the highlight is: low values produce a broad, diffuse sheen; high values create a sharp, glossy spot. Specular works with both flat and smooth shading, and is most effective with PNG output.

Diffuse only

Specular (`specular: 0.6, shininess: 32`)

Gamma Correction

By default (`gamma_correction: true`), colors are converted to linear space before shading and back to sRGB afterward. This produces physically accurate lighting: midtones brighten, dark areas gain detail, and specular highlights blend smoothly. Disabling it (`gamma_correction: false`) computes lighting directly in sRGB — faster, but produces harsher contrast and less natural results.

Without (`gamma_correction: false`)

With (default)

Fresnel / Rim Lighting

Fresnel rim lighting brightens edges where the surface curves away from the camera, making objects stand out from the background. Control with `fresnel: (intensity, power)` or just `fresnel: 0.6` for intensity only. Higher power gives a thinner rim. Works with both flat and smooth shading.

Without fresnel

With (`fresnel: 0.6`)

Multi-Light

By default, a single white directional light is used (from `light_dir`). The `lights` array lets you define multiple lights, each with a type, direction or position, color, and intensity. When `lights` is set, it overrides `light_dir`.

Each light has a `type`, a `vector`, a `color`, and an `intensity`. Three types share one schema, differing in what `vector` means:

- **`directional`** — parallel rays; `vector` is a direction (like sunlight).
- **`positional`** — a hard point light; `vector` is a world position, so shading is distance-dependent.
- **`area`** — a disk light at `vector` with radius `size` (see Area Lights); `size` applies to this type only.

Each light casts its own cast shadow when `shadows` is enabled (see the Cast Shadows section); the `ground_shadow` drop shadow instead uses a single direction — the first directional light, or `light_dir` as fallback.

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.19,
  ambient: 0.05,
  specular: 0.5,
  color: "#cccccc",
  lights: (
    (type: "positional",
     vector: (3, 3, 0),
     color: "#ff4444",
     intensity: 1.2),
    (type: "positional",
     vector: (-3, 2, 2),
     color: "#44ff44",
     intensity: 1.0),
    (type: "directional",
     vector: (0, 1, 0),
     color: "#4444ff",
     intensity: 0.5),
  )
)
```

Area Lights

Set a light's `type` to `area` and give it a `size` — its radius in world units — to make it a **disk area light** rather than an infinitesimal point. Two things follow, just like a real softbox: its shadow gains a penumbra that is sharp on contact and blurs with distance, and its specular highlight broadens and dims instead of forming a hard glint. `size` only applies to `area` lights — a `positional` light is always a hard point (see Multi-Light).

Soft shadows are computed with PCSS, so they need `shadows: (per_pixel: true)` (PNG only). The specular softening always applies.

`size: 0 (hard point)`

`size: 1.5`

`size: 4`

As `size` grows, the sharp specular glint spreads into a soft sheen and shadow edges blur into penumbra. `size` is a per-light property, so a scene can mix a small key light with a large, soft fill:

```
lights: ((type: "area", vector: (4, 7, 5), size: 1.5,
          color: "#fff", intensity: 2.4),),
shadows: (per_pixel: true),
```

Tone Mapping

When multiple bright lights, strong specular, or fresnel push color values above 1.0, the default behavior hard-clips them to white — creating flat, washed-out highlights. Tone mapping compresses these HDR values gracefully, preserving detail and color in bright areas. Two operators are available: `"reinhard"` (simple, neutral) and `"aces"` (filmic, higher contrast). Use `tone_mapping: (method: "aces", exposure: 1.5)` for full control, or just `tone_mapping: "aces"` for the method alone.

No tone mapping | **Reinhard** | **ACES**

Shading Models

The `shading` parameter selects the lighting model, to configure the lights behaviour in the scene.

Blinn-Phong (default)

Photorealistic diffuse + specular.

```
#render-obj(bunny, (
  up: (0, 1, 0),
  azimuth: 180, distance: 0.25,
  specular: 0.4,
  shading: "blinn-phong"
))
```

Normal Mapping

Maps surface normals to RGB.

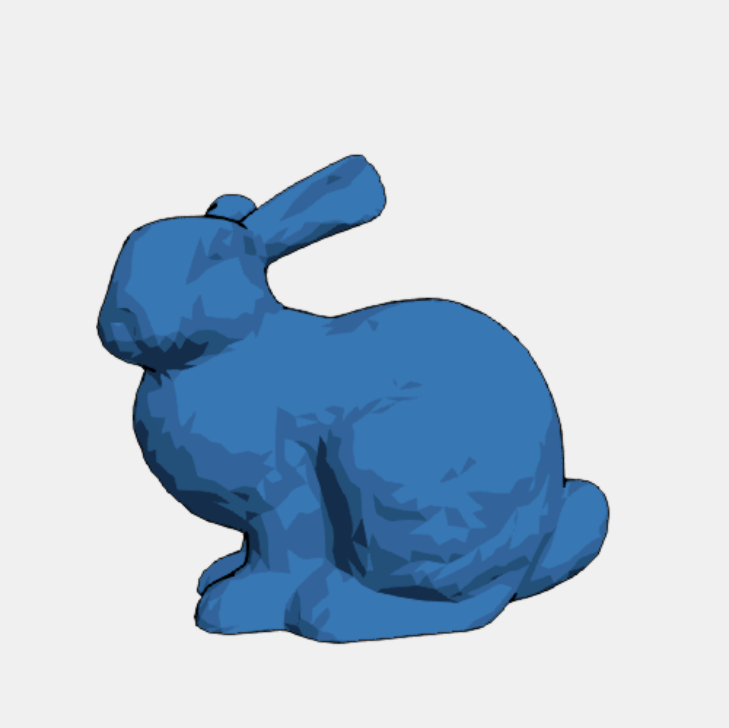
Flat

Face-normal shading. Disables `smooth` unless explicitly set.

Cel

Toon shading with discrete color bands.

Control steps with `cel_bands`.



Gooch

Warm-to-cool non-photorealistic shading.

Customize with `gooch_warm` and `gooch_cool`.

Subsurface Scattering

Maquette approximates subsurface scattering with a cheap, view-dependent hack rather than true volumetric light transport: the *Approximating Translucency* technique (Barré-Brisebois & Bouchard, GDC 2011) — a single dot product between the view direction and the back-facing light. It gives the warm glow of light passing through thin geometry (wax, skin, marble, leaves); back-lit areas glow with a color derived from the light and the model's base color. There's no real thickness sampling, so the glow is uniform rather than thickness-driven. Works with any shading model.

Without Subsurface Scattering

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  lights: (
    (type: "positional",
     vector: (-0.1, 0.14, -0.04),
     color: "#ff0000",
     intensity: 3.0),
  ),
)
```

Subsurface Scattering (back-lit bunny)

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  lights: (
    (type: "positional",
     vector: (-0.1, 0.14, -0.04),
     color: "#ff0000",
     intensity: 3.0),
  ),
  sss: (intensity: 4,
    power: 3.5,
    distortion: 0.2),
)
```

The `sss` dictionary has three knobs: **`intensity`** scales the overall glow; **`power`** sharpens its falloff — higher values let light show through only the thinnest parts (the ears here); and **`distortion`** wraps the transmitted light around the surface normal for a softer, broader spread.

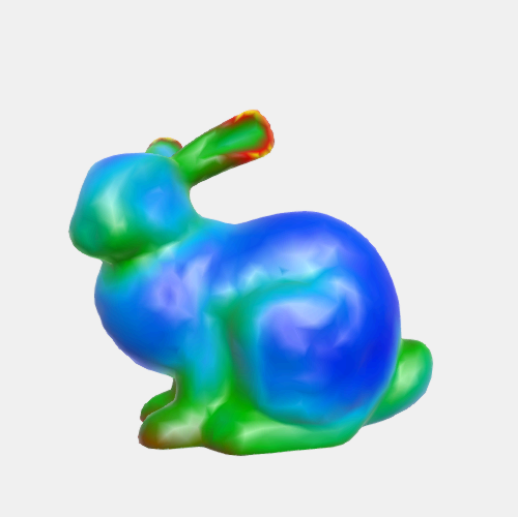
Color Mapping

Color mapping replaces the uniform model color with a gradient derived from geometric properties.

Curvature Map

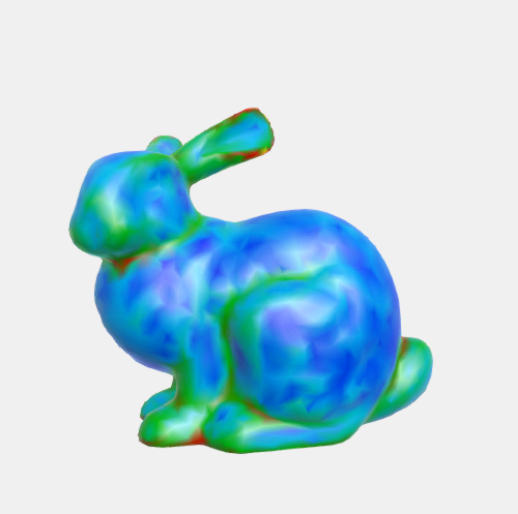
Colors vertices based on local surface curvature — the rate at which the surface bends. Low curvature (flat areas) maps to blue/dark colors, while high curvature (sharp edges, creases) maps to red/bright colors. Useful for quality inspection, identifying sharp features, or visualizing mesh topology.

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  ambient: 0.3,
  specular: 0.5,
  vertex_smoothing: 4,
  color_map: "curvature",
  width: 70%,
)
```



You might have noticed the appearance of a `vertex_smoothing` setting in the previous render. This parameter allows to smoothen the color distribution across vertices, otherwise the color looks flaky, as you can see with `vertex_smoothing: 0`:

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  ambient: 0.3,
  specular: 0.5,
  color_map: "curvature",
  vertex_smoothing: 0,
  width: 70%,
)
```



Overhang Map

Faces steeper than `overhang_angle` (relative to vertical) are highlighted in red, while supported faces remain green. This directly maps to where a 3D printer would need support structures.

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  ambient: 0.3,
  specular: 0.5,
  color_map: "overhang",
  overhang_angle: 45,
  width: 70%,
)
```

Scalar Function

Color vertices based on a user-defined mathematical function `f(x,y,z)`. The `scalar_function` expression is evaluated at each vertex position, producing scalar values that are automatically normalized to `[0, 1]` (min value → 0, max value → 1), then linearly interpolated through the `color_map_palette` color stops. If no palette is specified, the default blue → cyan → green → yellow → red gradient is used. Per-vertex colors are interpolated across triangle faces for smooth results.

The expression can use:

- **Variables:** `x`, `y`, `z` (vertex coordinates)
- **Constants:** `pi`, `e`, `tau`
- **Arithmetic:** `+`, `-`, `*`, `/`, `^` (power)
- **Comparison:** `<`, `>`, `<=`, `>=`, `==`, `!=` (return 0.0 or 1.0)
- **Logical:** `&&` (and), `||` (or), `!` (not)
- **Functions:**
  - Basic: `abs`, `sqrt`, `min`, `max`, `clamp`, `sign`, `pow`
  - Trigonometric: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
  - Hyperbolic: `sinh`, `cosh`, `tanh`
  - Exponential/Logarithmic: `exp`, `ln`, `log10`, `log2`
  - Rounding: `floor`, `ceil`, `round`, `fract`
  - Graphics: `step`, `smoothstep`, `mix`, `lerp`, `length`
  - Other: `mod`

Some examples:

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  ambient: 0.3,
  specular: 0.5,
  color_map: "scalar",
  scalar_function: "smoothstep(-0.1, 0.1, x)",
  width: 80%,
)
```

```
#render-obj(bunny,
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
  ambient: 0.3,
  specular: 0.5,
  color_map: "scalar",
  scalar_function: "sin(x*60)*cos(y*60) + sin(y*60)*cos(z*60) +
sin(z*60)*cos(x*60)",
  color_map_palette: (
    "#1a0533",
    "#6b2fa0",
    "#e85d75",
    "#ffcc33",
  ),
  width: 80%,
)
```

File Formats & Coloring

STL Per-face Color

Some binary STL files encode per-face colors in the attribute bytes using the RGB565 format. Maquette detects and renders these automatically — no config needed. When present, the `color` parameter is ignored in favor of the embedded colors.

```
#let colored = read("data/colored_cube.stl", encoding: none)

#render-stl(colored, projection: "isometric", width: 65%)
```

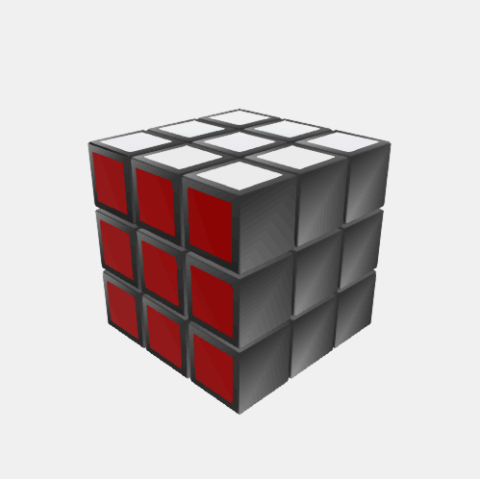
PLY Format

Meshes

Maquette handles PLY files in ASCII and binary (little/big-endian) formats — all three are parsed automatically. PLY can store colors in its format, allowing us to color the model directly. Enjoy this beautiful PLY-colored Rubik's cube generated with Blender.

```
#let rubi = read("data/rubi_blender.ply", encoding: none)

#render-ply(rubi,
  azimuth: 45,
  elevation: 25,
  distance: 9,
  width: 65%,
)
```



Point Clouds

PLY files can also contain clouds of points. 3D scanning apps usually allow to export in such a format. Enjoy my Rubik's cube scanned with the help of my iPad's LiDAR! Point clouds rendering are configurable with `point_size`, which will define the distance for points to be considered as neighbors for our k-NN to reconstruct the model. Auto-computed if zero.

```
#let rubi_scan = read("data/rubi_scan.ply", encoding: none)

#render-ply(rubi_scan,
  up: (0,1,0),
  elevation: 25,
  distance: 0.75,
  auto_fit: false,
  point_size: 0.03,
  width: 65%,
)
```

OBJ Material Coloring

OBJ files can reference materials via `usemtl` directives. Provide a `materials` map to assign hex colors to each material name.

We list `cube.obj`'s materials as follows:

```
$ grep usemtl cube.obj
usemtl face1
usemtl face2
usemtl face3
usemtl face4
usemtl face5
usemtl face6
```

Then colorize material-wise:

```
#let obj-cube = read("data/cube.obj")

#render-obj(obj-cube,
  camera: (3,3,3),
  materials: (
    face4: "#2ecc71",
    face5: "#222222",
    face6: "#ff2222",
  ),
)
```

OBJ Groups

Group Highlight

OBJ files with `g` or `o` directives define named groups. The `highlight` map assigns custom styling to specific groups.

Groups not listed keep their default appearance.

Available Attributes

When using the full object syntax, all attributes are optional:

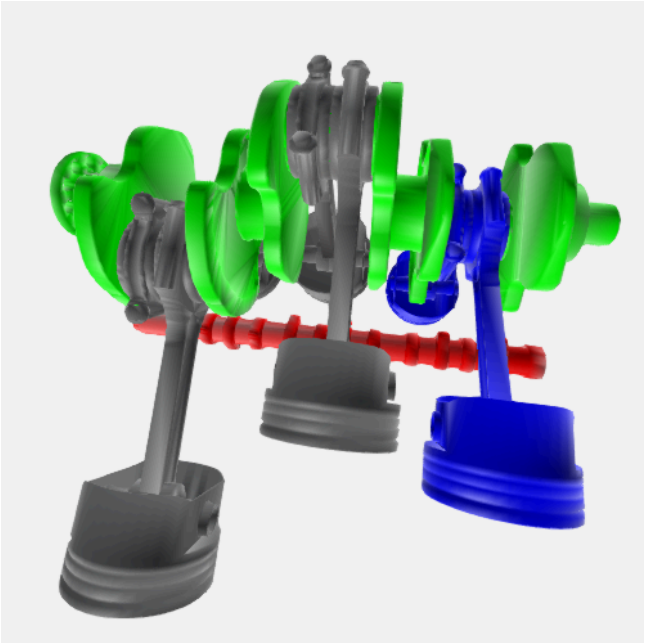
```
"highlight": (
  "GroupName": "#c0ffee",    // Simple: just a color string
  "AnotherGroup": (          // Advanced: full appearance object
    "color": "#ff0000",      // Hex color (overrides global color)
    "specular": 0.8,         // Specular intensity 0-1 (overrides global)
    "shininess": 64,         // Specular exponent (overrides global)
    "ambient": 0.3,          // Ambient light 0-1 (overrides global)
    "stroke": "#000000",     // Triangle edge stroke color
    "stroke_width": 1.0,     // Triangle edge stroke width
    "opacity": 0.5,          // Transparency 0-1 (0=invisible, 1=opaque)
    )
)
```

We can list the groups as follows:

```
$ grep "g " crankshaft.obj
g Model__Piston_F
g Model__Head
g Model__Piston_E
g Model__Head_2
g Model__Piston_D
g Model__Head_3
g Model__Piston_C
g Model__Head_4
g Model__Piston_B
g Model__Head_5
g Model__Piston_A
g Model__Head_6
g Model__Crankshaft
g Model__Camshaft
```

Here's an example of a crankshaft with several parts defined by groups in the `.obj` file format:

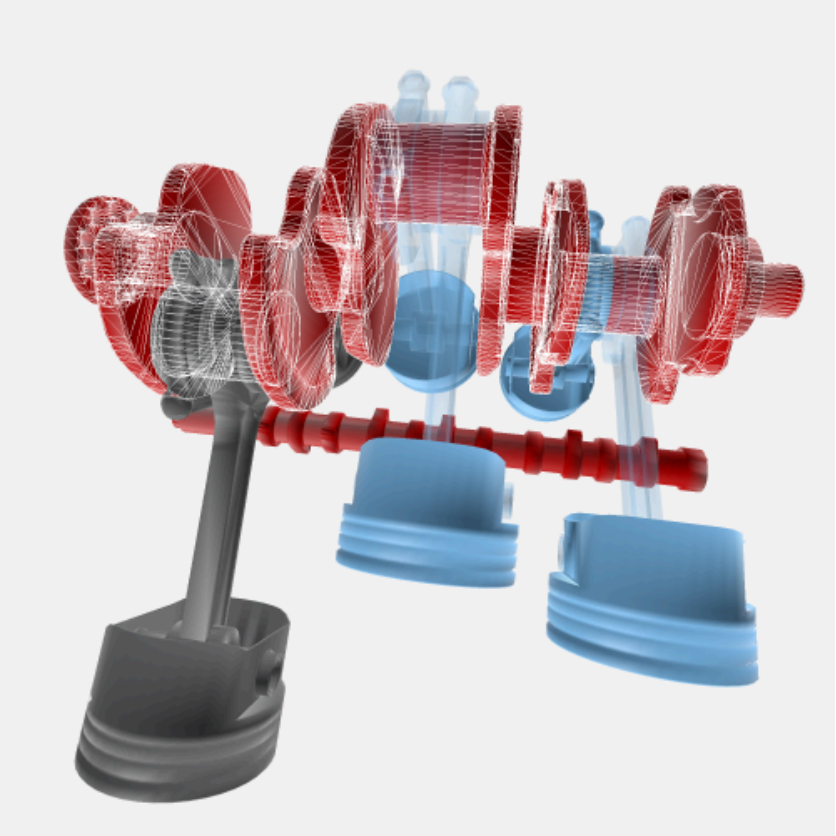
```
#align(center,
render-obj(crankshaft,
  camera: (-100, -100, 500),
  up: (0, -1, 0),
  zoom: 1.25, pan: (0, 0.08),
  color: "#777777",
  highlight: (
    Model__Camshaft: "#ff0000",
    Model__Crankshaft: "#00ff00",
    Model__Piston_B: (color: "#0000ff"),
  ),
  width: 87%,
))
```



Per-group appearance

Instead of a plain color, pass a dictionary with specific appearance overrides to selectively change parts appearance.

```
#render-obj(crankshaft,
  camera: (-100, -100, 500),
  up: (0, -1, 0),
  zoom: 1.25, pan: (0, 0.08),
  color: "#777777",
  antialias: 4,
  highlight: (
    Model__Crankshaft:
      (color: "#cc0000",
       stroke: "#ffffff",
       stroke_width: 0.5),
    Model__Piston_A: (color: "#88ccff", opacity: 0.3),
    Model__Piston_C: (color: "#88ccff", opacity: 0.3),
    Model__Piston_D: (color: "#88ccff", opacity: 0.3),
  ),
)
```

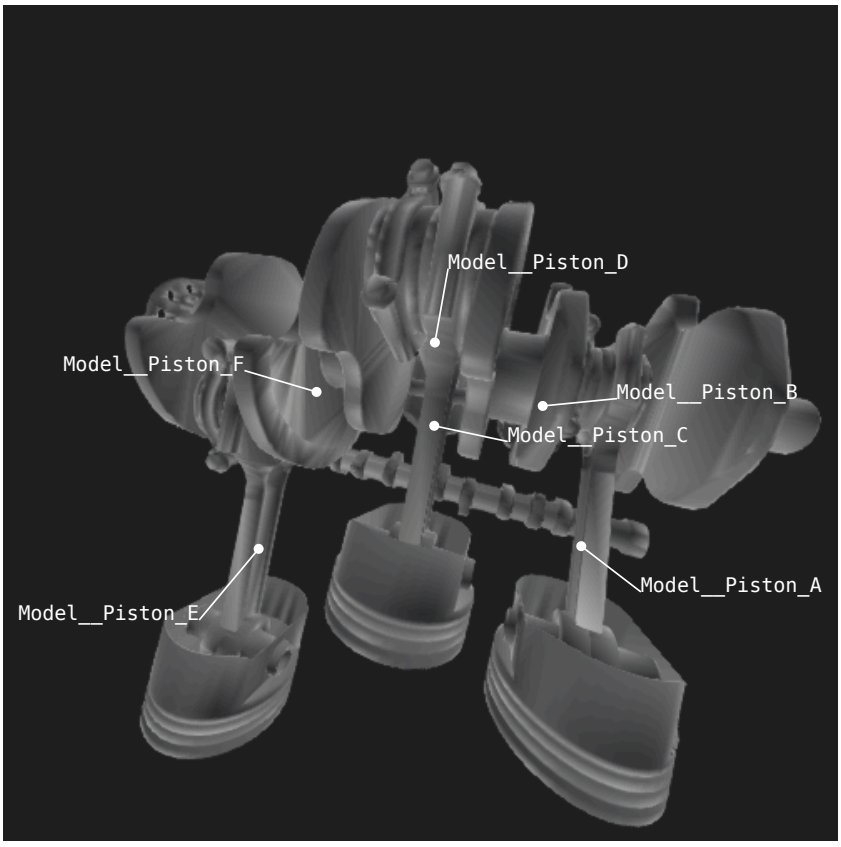


Annotations

Annotate OBJ groups by drawing a leader line from each group's centroid to a text label.

Pass `annotations: true` to label all groups with default styling, or pass an object to customize. The `groups` field filters to specific groups; `color`, `font_size`, and `offset` control appearance.

```
#render-obj(crankshaft,
  camera: (-110, -90, 380),
  up: (0, -1, 0),
  color: "#777777",
  annotations: (
    groups:
      ("Model__Piston_A",
       "Model__Piston_B",
       "Model__Piston_C",
       "Model__Piston_D",
       "Model__Piston_E",
       "Model__Piston_F"),
    color: "#ffffff",
    font_size: 12,
    offset: 45,
  ),
  background: "#222222"
)
```



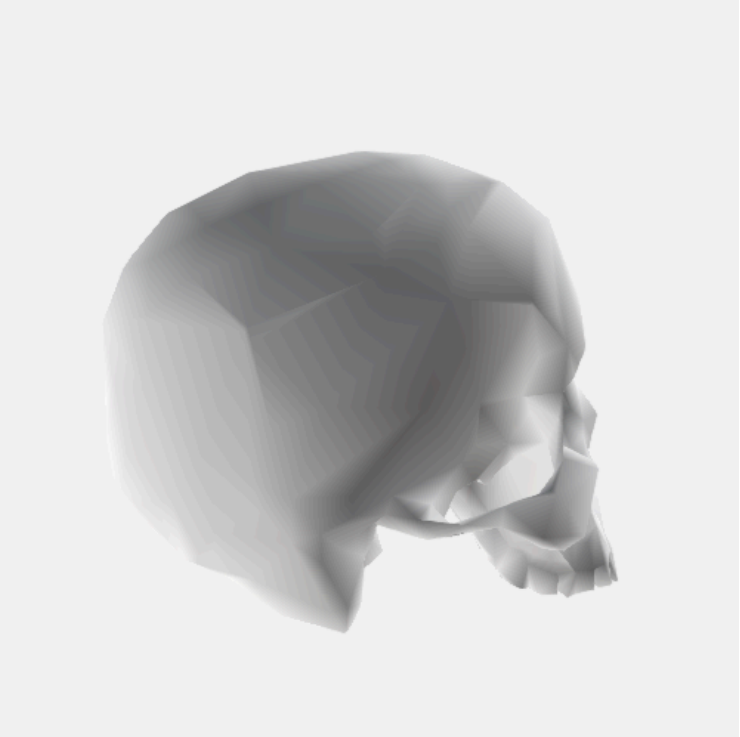
Render Modes

Solid (default)

Nothing new, since it's the default mode we saw earlier, but it allows me to introduce this beautiful low poly `brain_skull.obj`.

```
#let skull-brain = read("data/brain_skull.obj")

#render-obj(skull-brain,
  azimuth: 270,
  up: (0,1,0),
  distance: 200,
  color: "#e8e8e8",
)
```



X-Ray Mode

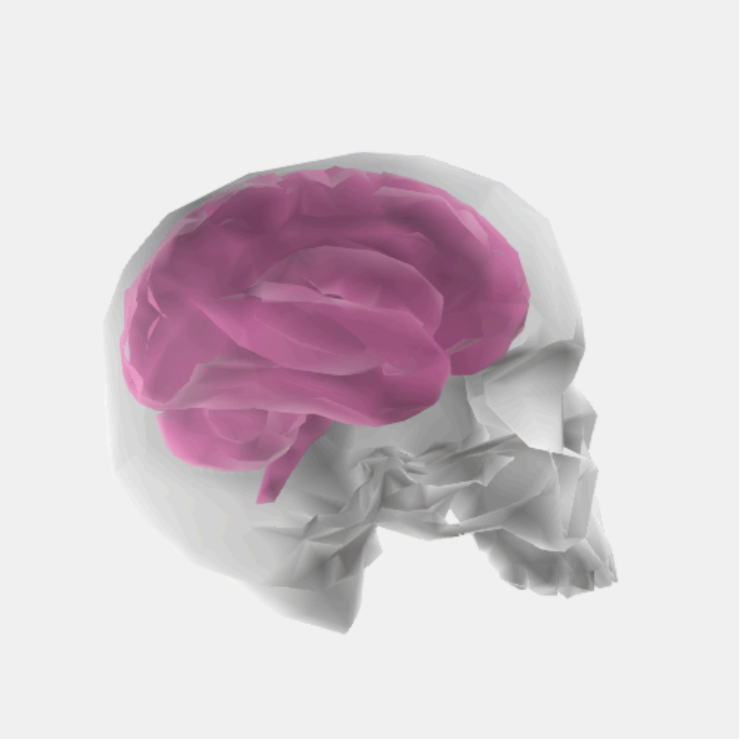
(`mode: "x-ray"`) creates transparent front-facing surfaces. Back-facing surfaces remain fully opaque.

Adjust transparency with `xray_opacity` (default 0.1 opacity).

Ideal for examining joints, internal components, nested geometry, and embedded models.

Our previous skull model now unveils its inner brain!

```
#render-obj(skull-brain,
  azimuth: 270,
  up: (0,1,0),
  distance: 200,
  highlight: (
    "Skull": (color: "#e8e8e8"),
    "Brain": (color: "#ee69b4"),
  ),
  xray_opacity: 0.3,
  mode: "x-ray",
)
```



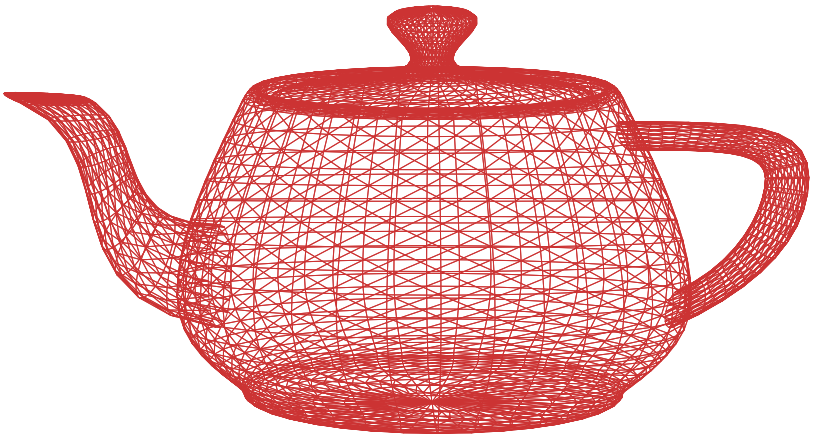
Useful when the model has no groups to distinguish.

Wireframe

Wireframe mode draws every triangle edge without fill or back-face culling, showing the full mesh topology. Control appearance with `wireframe: (color, width)`.

It's a great occasion to showcase SVG output, no rasterization artifacts and kind-of infinite zooming allowed, thanks to vectors.

```
#render-obj(teapot,
  up: (0, 1, 0),
  distance: 8,
  auto_fit: false,
  background: "#ffffff",
  wireframe: (color: "#cc3333", width: 0.1),
  mode: "wireframe",
  format: "svg",
  width: 55%,
)
```



Solid + Wireframe

Combines solid shading with wireframe edges overlaid on top. Useful for visualizing mesh density and triangle distribution while still seeing the shaded surface. Configure edge appearance with `wireframe: (color, width)`.

Here `antialias: 4` should be set, wireframe's strokes benefit from antialiasing.

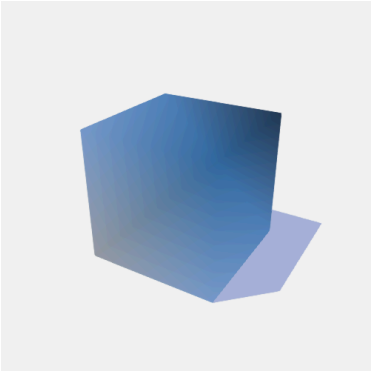
```
#render-obj(teapot,
  up: (0, 1, 0),
  distance: 8,
  antialias:4,
  mode: "solid+wireframe",
  wireframe: (width: 0.3),
  width: 80%,
  zoom: 1.2,
)
```

Shadows

Ground Shadow

A ground shadow is cast by projecting every triangle onto the ground plane along the light direction. Pass `ground_shadow: true` for defaults, or customize:

```
#render-stl(cube,
  camera: (3, 2, 2),
  light_dir: (1, -1, 3),
  ground_shadow: (opacity: 0.35, color: "#2244aa"),
  width: 50%,
)
```



Cast Shadows

PNG ONLY `shadows` renders true **self-shadowing** — every part occluding every other, computed with a depth map per light.

Option	Default	What it does
`per_pixel`	`false`	Sample shadows per fragment instead of per vertex — crisp edges on low-poly and CAD models. PNG only, ~2.5× the cost, and required by `light_size` and `color` below.
`light_size`	`0`	Light radius in world units. Any value `> 0` enables the PCSS soft shadows described above — sharp where parts touch, blurring with distance.
`color`	`""`	Hex tint for the shadowed regions instead of darkening toward neutral grey (e.g. a cool blue).
`strength`	`1.0`	How dark shadows go, from `0` (none) to `1` (removes all direct light).
`softness`	`1`	PCF (percentage-closer filtering) blur radius, in texels: `0` = hard edges, `1` = 3×3, higher = softer everywhere.
`resolution`	`512`	Shadow-map size per light (res × res texels). Higher is sharper but slower to build.
`omni`	`false`	Render six cube-map faces so a positional light **inside** the geometry casts in every direction. ~6× the cost.
`bias`	`0.0008`	Constant depth-compare bias — the baseline fix for shadow acne (self-shadowing speckle).
`normal_bias`	`2.0`	Offsets the sample along the surface normal (in texels); the primary acne fix.
`slope_bias`	`1.0`	Adds extra bias on surfaces lit at a grazing angle, where acne is worst.

`cast_shadow` is a per-light option, set inside the `lights` array (see Multi-Light), not a `shadows` one.

`shadows: false (default)`

`shadows: (per_pixel: true, light_size: 8, …)`

Post-Processing

Antialiasing

PNG ONLY The `antialias` parameter is the single antialiasing control for PNG output — SVG is vector, so it needs none. `0` turns it off; `1` (the default) runs a fast **FXAA** edge-smoothing pass with no supersampling; `2` renders at 2×2 the resolution and downsamples for smoother edges; `4` gives the highest quality (`3` and `4` are equivalent — both render at 4× internally).

As a rule of thumb FXAA (`antialias: 1`) is enough for most renders — reach for `antialias: 4` with wireframe or stroke, where straight edges benefit most.

No antialiasing (`antialias: 0`)

FXAA (`antialias: 1`, default)

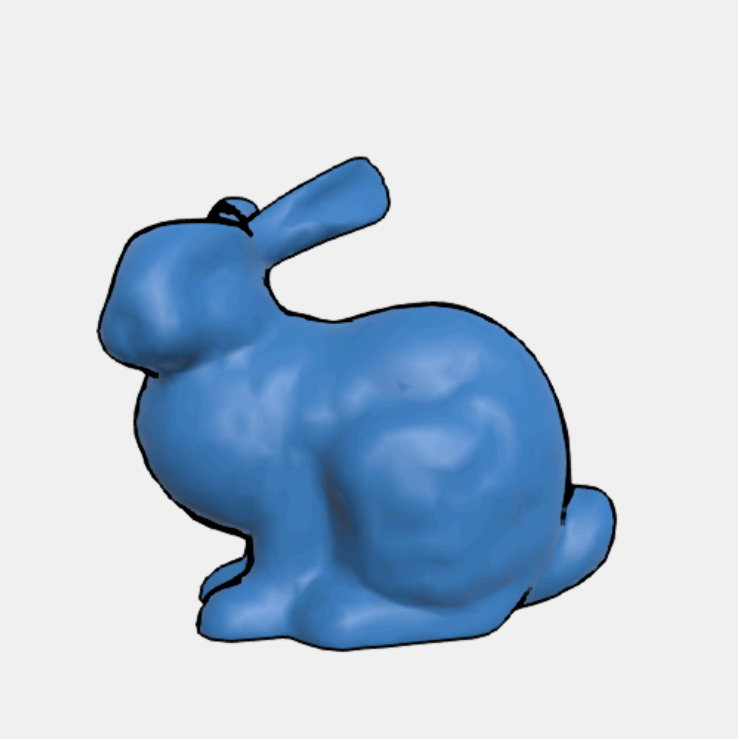
SSAA (`antialias: 2`)

4× supersampling (`antialias: 4`)

Silhouette Outlines

Draws bold edges where front-facing and back-facing triangles meet, producing a clean silhouette contour.

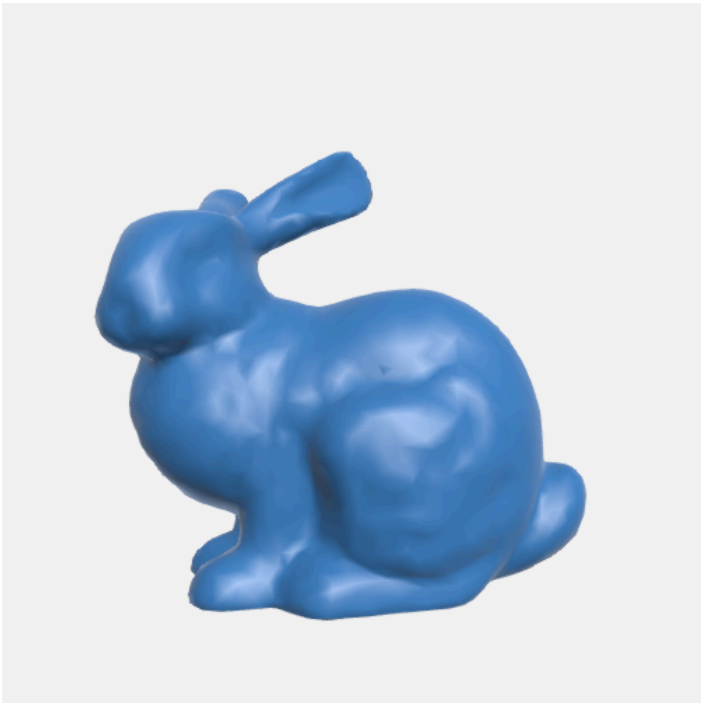
```
#render-obj(bunny,
  outline: (color: "#000000", width: 5),
  up: (0, 1, 0),
  azimuth: 180,
  distance: 0.25,
)
```



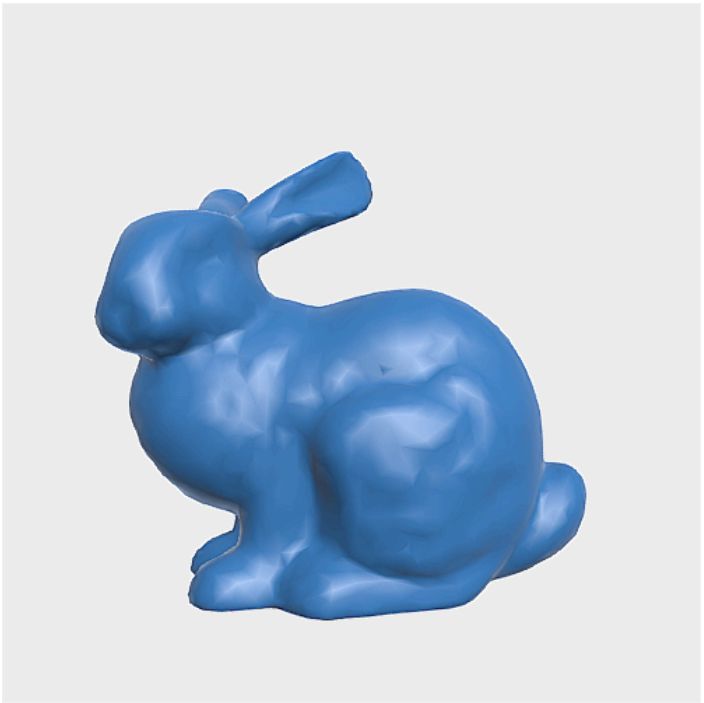
Sharpening

PNG ONLY Sharpening enhances edge contrast using a 3×3 unsharp mask. Pass `sharpen: true` for default strength (0.5), or customize with `sharpen: (strength: N)`. Higher values produce a more pronounced effect.

Without



Sharpen (strength: 2)



Bloom & Glow

PNG ONLY Bloom makes bright areas bleed light outward. Glow creates a uniform aura around the model's silhouette. Use `bloom: true` / `glow: true` for defaults, or customize:

```
bloom: (threshold: 0.8, intensity: 0.3, radius: 10)
glow: (color: "#ffffff", intensity: 0.5, radius: 15)
```

Bloom

Glow

Ambient Occlusion

PNG ONLY Ambient Occlusion adds realistic contact shadows (⚠️ at the cost of increased processing time) in crevices and areas where surfaces are close together, simulating how indirect light is blocked in tight spaces. SSAO computes occlusion by sampling the depth buffer after rasterization. Configure with `ssao: true` for defaults, or customize:

```
#render-obj(crankshaft,
  ssao: (samples: 16, radius: 0.5, bias: 0.025, strength: 1.0),
)
```

Without SSAO

With SSAO

Effects

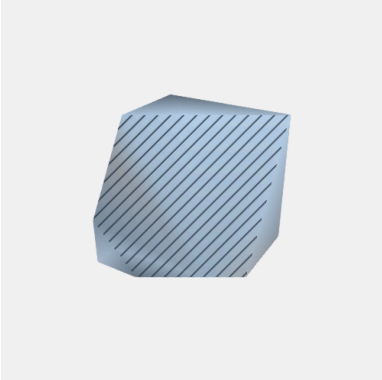
Clipping

Slice the model with a plane to cut part of it away — for section drawings or to reveal internal geometry. `clip` takes either an explicit world-space plane `(a, b, c, d)`, keeping the `ax + by + cz + d >= 0` half, or a dictionary that positions the plane for you.

Wrapping the plane in a dict lets you add `cap: false`, which leaves the cross-section open so you can see inside — here an explicit plane opens the skull to reveal the brain:

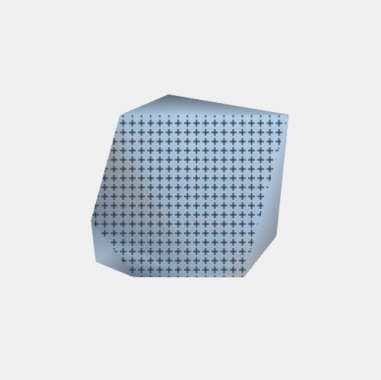
```
#render-obj(skull-brain,
  azimuth: 220,
  up: (0, 1, 0),
  distance: 180,
  highlight:
    ("Skull": (color: "#e8e8e8"),
     "Brain": (color: "#ff69b4")),
  clip: (plane: (2, -1, 0, 1), cap: false),
  cull_backface: false,
)
```

Cap it instead (the default) and the cut face can be **hatched** for an engineering-style section view. Three patterns — parallel `"lines"`, a `"cross"` grid, and discrete `"crosses"` — each honour `angle`, `spacing`, `width`, and `color` (the whole pattern rotates with `angle`), in SVG and PNG alike:



`style: "lines", angle: 45`

`style: "cross", angle: 30`



`style: "crosses", angle: 0`

The dict form positions the plane and controls the cut:

- **`from: "camera"`** squares the plane to the view direction, so the slice always faces the viewer whatever the angle (or use `axis: "x"/"y"/"z"`, `normal: (x, y, z)`, or a raw `(a, b, c, d)` plane).
- **`depth`** sets how deep to cut — `0` (near face) to `1` (far); `distance` sets it in world units instead.
- **`keep`** chooses which half to keep — `"far"` (default) or `"near"`.
- **`cap`** closes the cross-section with a flat face (default `true`); `false` reveals hollow interiors, as in the skull above.
- **`hatch`** draws section lines over the cap — `true` for defaults, or a dict of `style` (`"lines"`/`"cross"`/`"crosses"`), `angle`, `spacing`, `width`, and `color`. Needs `cap: true`.

Exploded View

Move model parts outward from the model center. Very useful to showcase different parts of a system in mechanics.

For OBJ files with `g` or `o` groups, each group is treated as a separate component.

```
#render-obj(crankshaft,
  up: (0, -1, 0),
  camera: (-200, -200, 500),
  color: "#555555",
  explode: 0.8,
)
```

For PLY, STL files or OBJ files without groups, connected components are detected automatically using shared edges (union-find). Each component is then offset by `explode * (component_centroid - model_center)`.

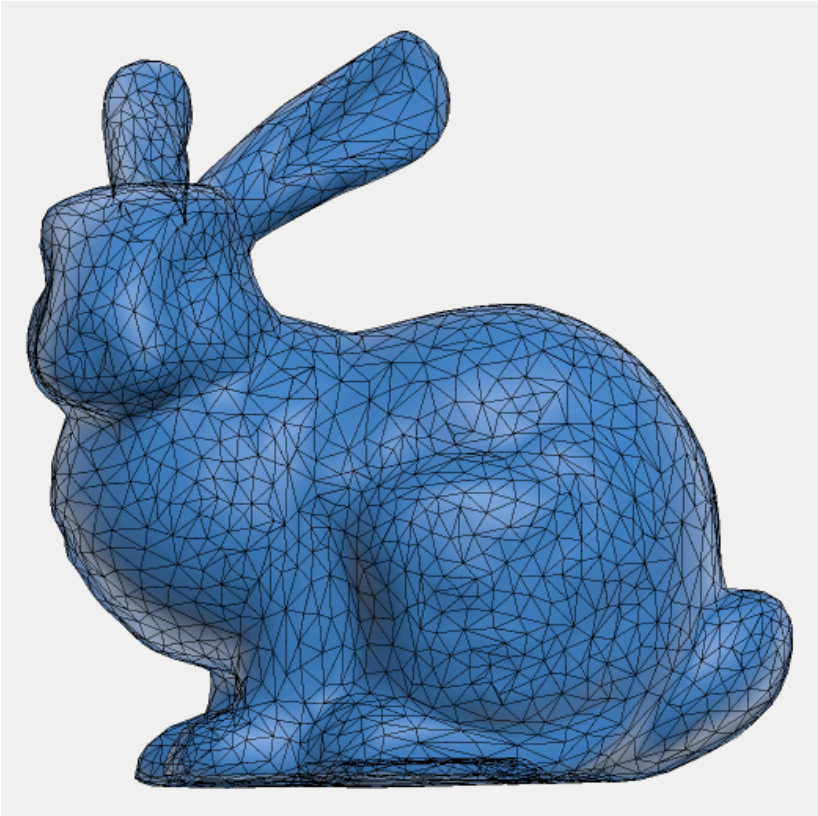
This is the case for this exploded teapot.

```
#render-obj(teapot,
  camera: (0, 2, 5),
  up: (0, 1, 0),
  explode: 0.5,
)
```

Decimation

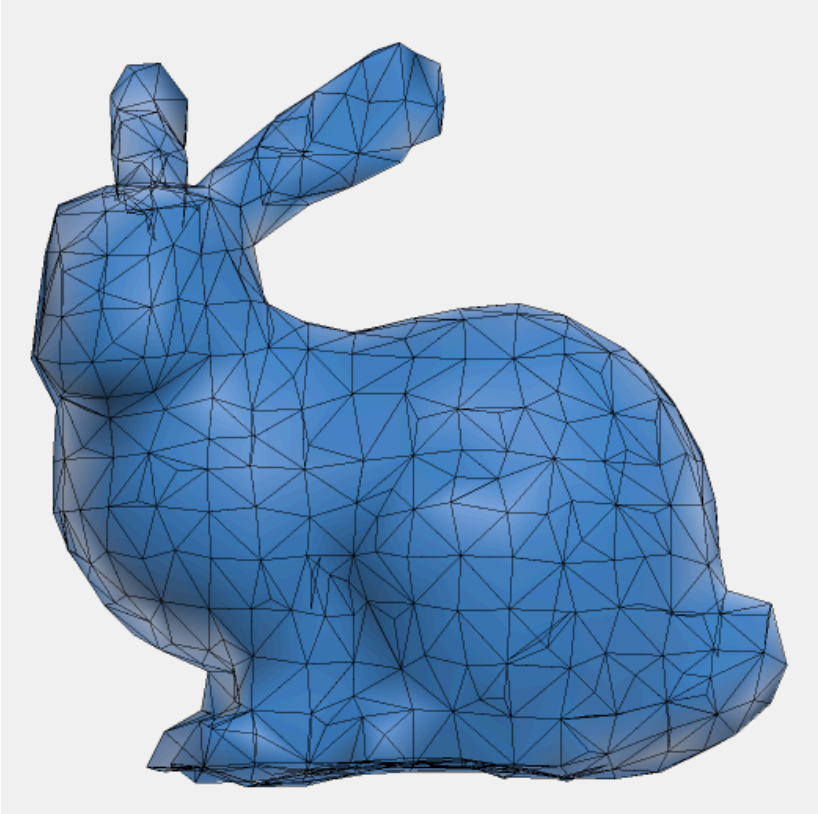
Reduce a mesh's triangle count with grid vertex clustering — a fast, format-agnostic mesh simplification that applies identically to STL, OBJ and PLY. It is handy for shrinking dense scans or high-poly exports so they render (and embed in your PDF) faster. It pairs well with the default smooth shading, which re-derives vertex normals and softens the faceting.

The `decimate` strength runs from `0` (off, default) to `1` (most aggressive): a uniform grid is laid over the model, vertices sharing a cell collapse into one, and higher values use a coarser grid that merges more detail. The wireframe overlay below makes the thinning topology visible.



Original (`decimate: 0`)

Vertices: **2503**
Triangles: **4968**



`decimate: 0.75`

Vertices: **618**
Triangles: **1288**

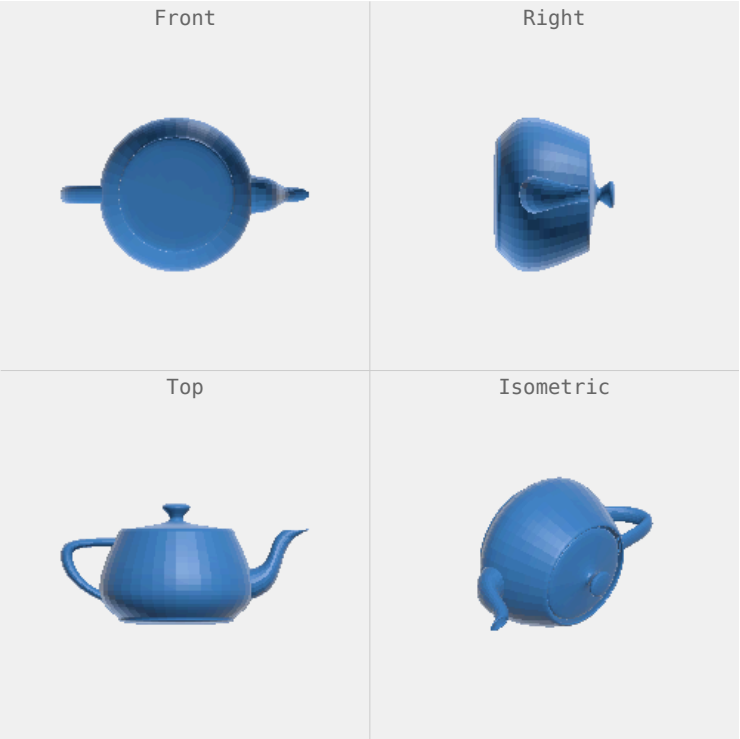
Fewer triangles means faster rendering, and the default smooth shading re-derives vertex normals afterward, so moderate decimation stays visually clean.

Multi-View

Multi-View Grid

Render multiple named views in a single image, similar to an engineering drawing sheet. Available views are `"front"`, `"back"`, `"left"`, `"right"`, `"top"`, `"bottom"`, and `"isometric"`. The renderer arranges them in a grid and labels each cell.

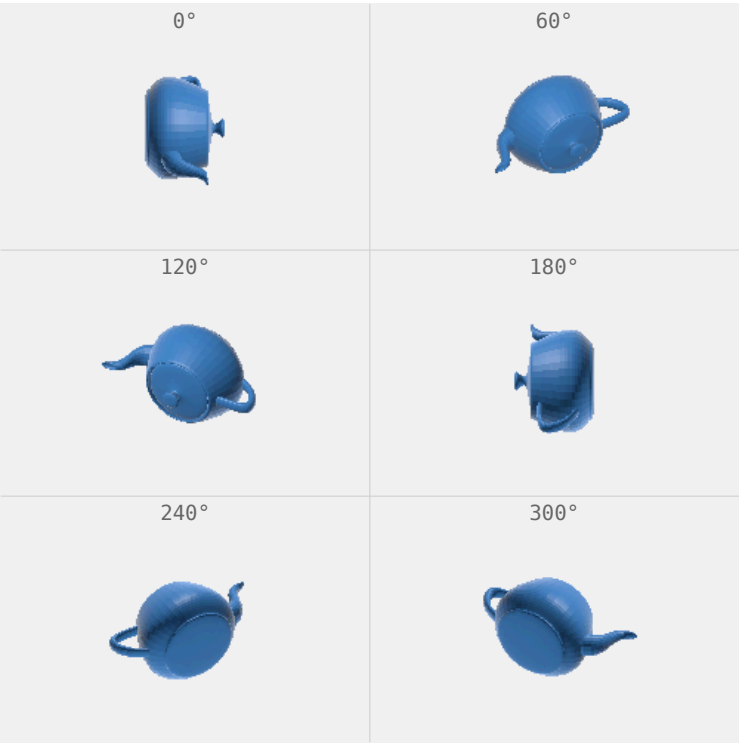
```
#render-obj(teapot,
  views: ("front", "right", "top", "isometric"),
  grid_labels: true,
  cull_backface: false,
)
```



Turntable

Automatically generates a grid of views evenly spaced around the model at a fixed elevation angle. Use `turntable: (iterations: 6, elevation: 40)` or just `turntable: 6` for the number of views. View labels showing the azimuth angle are displayed by default; set `grid_labels: false` to hide them.

```
#render-obj(teapot,
  turntable: (iterations: 6, elevation: 40),
  grid_labels: true,
  cull_backface: false,
)
```

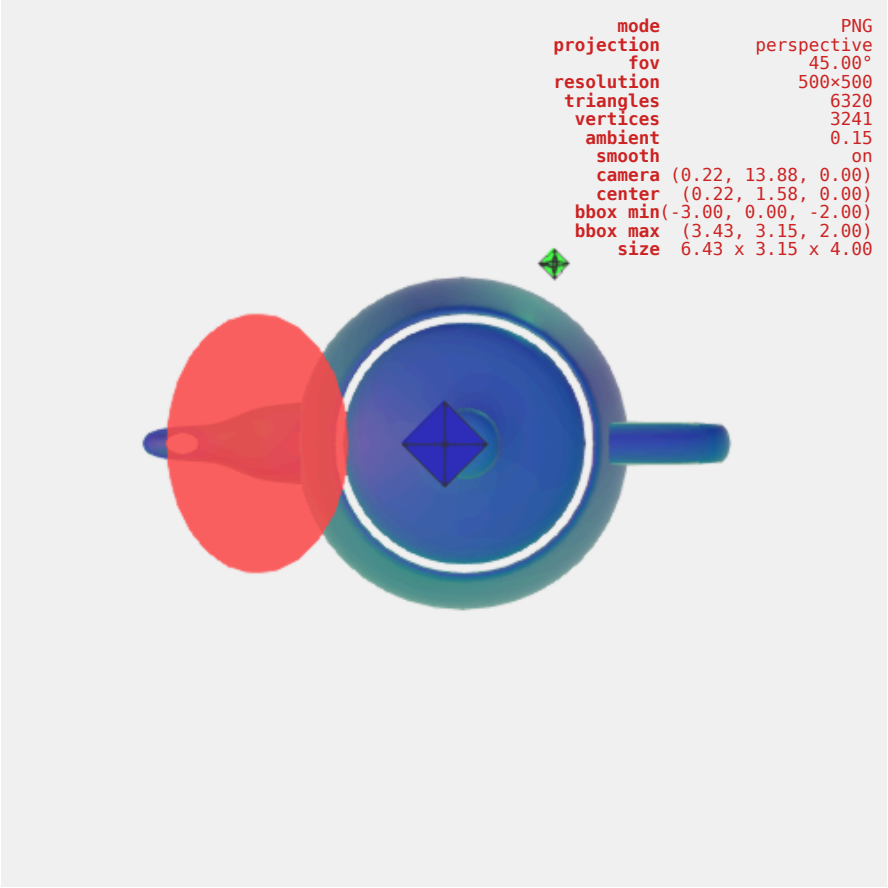


Debug

`debug: true` overlays model metadata (triangle count, bounding box, camera position) directly on its canvas. The overlay text is drawn in `debug_color` (default `#cc2222`) — set it to keep the labels legible against your model or background.

It also renders lights as octahedrons of the color they emit, to allow placing lights seamlessly around your model. Area lights (those with a `size`) are drawn instead as a disk of that radius, facing the model, so you can gauge their extent.

```
#render-obj(teapot,
  debug: true,
  lights: (
    (type: "area",
     vector: (2, 4, 0),
     size: 1.2,
     color: "#ff4444",
     intensity: 1.2),
    (type: "positional",
     vector: (-1, 2, 2),
     color: "#44ff44",
     intensity: 1.0),
    (type: "directional",
     vector: (0, 1, 0),
     color: "#4444ff",
     intensity: 0.5),
  ),
)
```



Model Info & Measurements

`get-stl-info`, `get-obj-info`, and `get-ply-info` return a dictionary of the model's metadata and geometric measurements, all in the file's own units. Alongside `triangles`, `vertices`, the bounding box (`bbox_min`/`bbox_max`/`bbox_center`/`bbox_radius`), and the resolved `camera`/`center`/`projection`/`fov`, it reports:

Field	What it is
`size`	Bounding-box dimensions `(dx, dy, dz)`.
`surface_area`	Total triangle area (exact).
`volume`	Enclosed volume — exact for a closed, consistently-wound mesh; approximate for open or non-manifold ones.
`centroid`	Centre of mass `(x, y, z)` (volume-weighted).

Because it's a plain dictionary, you can drive labels, callouts, or camera framing from real geometry. Here it is run on the bunny:

```
#let info = get-obj-info(bunny)
#grid(columns: 2, column-gutter: 1em, row-gutter: 0.3em, align: (right, left),
  ..for (key, val) in info.pairs() { (strong(key), repr(val)) })
```

triangles	4968
vertices	2503
bbox_min	(-0.0944, 0.0333, -0.0617)
bbox_max	(0.0608, 0.187, 0.0587)
bbox_center	(-0.0168, 0.1102, -0.0015)
bbox_radius	0.1247
size	(0.1552, 0.1537, 0.1204)
surface_area	0.0565
volume	0.0008
centroid	(-0.0208, 0.0859, 0.011)
camera	(-0.0168, 0.4842, -0.0015)
center	(-0.0168, 0.1102, -0.0015)
projection	"perspective"
fov	45.0

Models Credits

- Utah teapot — Stanford
- Stanford bunny — Stanford
- Crankshaft with pistons — CGTrader
- Low-poly brain — Sketchfab
- Low-poly skull — Printables
- Rubik's cubes: Blender generated & LiDAR scanned by myself

Contributing

Bug reports, feature requests, issues, new feature ideas are welcome on GitHub.