Povrayst

Declarative raytracing in Typst.

version 0.1.0 · April 2026

github.com/bernsteining/povrayst · typst.app/universe/package/povrayst

Contents

1. Introduction 2
2. Caching and iteration 2
3. Scene Description Language 2
4. Getting started 3
5. Config reference 4
6. Resolution & quality 5
    6.1. `width` / `height` 5
    6.2. `quality` 5
7. Antialiasing 5
8. Recipes 6
9. Transparent background 6
10. Multi-file scenes (`includes`) 6
11. Typst-driven scene parameterisation 7
12. Tracing depth & advanced options 7
    12.1. `max-trace-level` 7
    12.2. `extra` (escape hatch) 8
13. Examples 8
    13.1. Constructive solid geometry 8
    13.2. Quaternion Julia fractal 10
    13.3. Clebsch diagonal cubic 11
    13.4. Schoen gyroid 12
    13.5. Lissajous knot 8_{21} 13
    13.6. Hopf fibration 14
    13.7. Advanced lighting 15
14. Debugging 16
15. Further reading and other examples 17
16. Render times 17
17. Conclusion 17

1. Introduction

POV-Ray (*Persistence of Vision Raytracer*) is a free, declarative ray tracer first released in 1991. There is no interactive editor: a scene is a plain text file describing camera, lights, geometry, and surfaces, and the renderer turns it into a pixel-perfect image. The same `.pov` source produces the same output on any platform and any version — scenes diff cleanly, parameterise easily, and stay reproducible.

What makes POV-Ray useful for mathematical and scientific figures is the breadth of primitives: implicit algebraic surfaces (`cubic`, `quartic`, `poly` up to degree 15), quaternion fractals (`julia_fractal`), swept tubes (`sphere_sweep`), photon caustics, and `function { ... }` expressions for arbitrary procedural patterns and isosurfaces. The gallery at the end of this document — Clebsch cubic, Hopf fibration, Lissajous knot, gyroid, caustics — is a few dozen lines of SDL each, no external assets. Povrayst wraps POV-Ray 3.8 as a Typst WebAssembly plugin so all of that renders inline when you compile.

2. Caching and iteration

Typst memoises `render()` by its arguments. Identical calls across compiles return the cached PNG without re-running the plugin — change the scene string or any kwarg and it re-renders; otherwise it's free. This makes incremental documentation editing very fast.

Run `typst watch doc.typ` to pick up changes on save. A cold first compile runs every scene once (typically a few seconds to a couple minutes each, depending on complexity); subsequent edits to prose, layout, or unrelated sections complete in milliseconds.

For expensive renders you iterate on rarely (a high-resolution cover, say), compile once to a separate `.typ` that writes a PNG to disk, then reference that PNG via `image()` in your main document — the logo at the top of this PDF is produced that way, since I needed the PNG for the `README.md`.

3. Scene Description Language

A minimal POV-Ray scene has a `camera`, at least one `light_source`, and one or more objects:

```povray
camera { location <0, 2, -5> look_at 0 angle 40 }
light_source { <4, 6, -4> color rgb 1.2 }
sphere { 0, 1 pigment { rgb <1, 0.4, 0.15> } }
```

Objects carry a `pigment` (colour) and optionally a `finish` (specular/diffuse/reflection), an `interior` (index of refraction, dispersion), and a `normal` (surface bump).

Pigments and normals can be procedural patterns — `checker`, `gradient`, `spherical`, `marble`, `wood`, `granite`, `crackle`, or arbitrary `function { x*x + y*y + sin(z) }` expressions — so most textures need no image files at all.

`#declare`, `#while`, `#macro`, and arithmetic on every numeric parameter let you parameterise scenes programmatically; the gallery's Hopf and gyroid examples use this to generate fibres and isosurface functions.

Have a look at the Scene Description Language's documentation for more information.

4. Getting started

The package exports two functions, `pov` and `render`. Both accept a POV-Ray scene plus the same keyword arguments (full reference in Section 5). They differ only in how the scene is supplied:

- `pov(scene, ..kwargs)` accepts the scene as a **string** or as a fenced `povray` **raw block**. Use it for inline scenes.
- `render(scene, ..kwargs)` accepts the scene as a **string only**. Use it for scenes loaded from `.pov` files via `read()`.

Internally `pov` extracts the text from a raw block (or passes a string straight through) and forwards to `render`.

```typst
#import "../pkg/povray.typ": pov, render

// Inline scene with `pov`:
#pov("camera { location <0, 2, -6> look_at 0 angle 35 }
      light_source { <4, 6, -4> rgb 1.2 }
      sphere { 0, 1 pigment { rgb <0.90, 0.40, 0.15> } }",
  width: 480, height: 360)

// Scene from a file with `render`:
#render(read("scene.pov"), width: 480, height: 360)
```

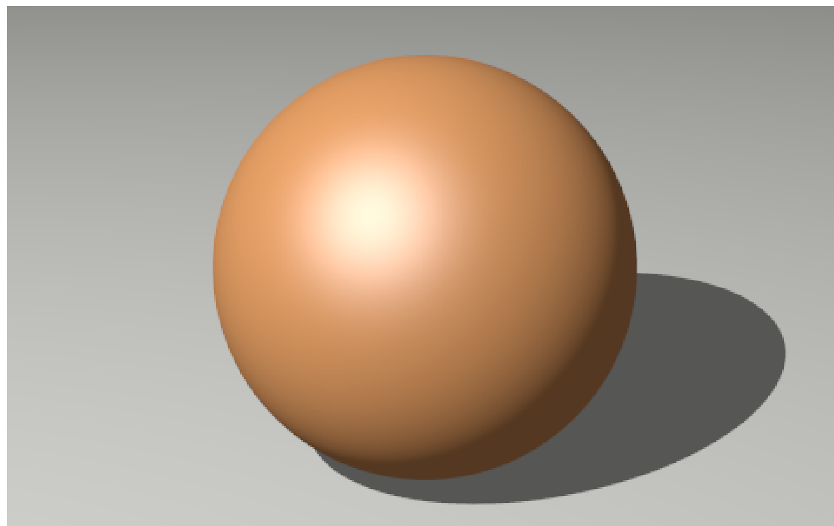


Figure 1: The scene above rendered.

For really terse one-shots, install a show rule so that every fenced `povray` block becomes a render in place:

```typst
#show raw.where(lang: "povray"): pov
```

After this, any fenced `povray` block in your document renders directly, with no `#pov(...)` wrapper. Useful to keep syntax highlighting and avoid writing code in a raw string.

````povray
```povray
camera { location <0, 2, -6> look_at 0 angle 35 }
light_source { <4, 6, -4> rgb 1.2 }
sphere { 0, 1 pigment { rgb <0.90, 0.40, 0.15> } }
````
````

````
```
````

5. Config reference

Every keyword argument to `render()`, its default, and the POV-Ray option it maps to. The option names in the right column come verbatim from POV-Ray's `.ini` / command-line reference — see the official command-line options for the full semantics of each flag. Also check POV-Ray's documentation.

kwarg	default	POV-Ray option	effect
Includes			
`includes`	`(:)`	`Typst-side #include expansion`	dictionary `{name: content}` spliced before parsing
Output resolution			
`width`	`800`	`Width`	image width in pixels
`height`	`600`	`Height`	image height in pixels
Quality			
`quality`	`9`	`Quality (0–11)`	`0` ambient only, `2` +shadows, `5` +reflection, `9` full radiosity
Antialiasing			
`antialias`	`true`	`Antialias=on/off`	master switch for adaptive supersampling
`aa-threshold`	`0.3`	`Antialias_Threshold`	colour delta that triggers extra samples; lower = smoother
`aa-method`	`2`	`Sampling_Method (1/2/3)`	`1` fixed grid, `2` adaptive recursive, `3` generic oversampling (3.8+)
`aa-depth`	`3`	`Antialias_Depth (1–9)`	recursion depth; up to depth^2 samples/pixel
`aa-jitter`	`true`	`Jitter=on/off`	randomise sub-pixel positions to break up aliasing
`aa-jitter-amount`	`none`	`Jitter_Amount`	jitter magnitude 0.0–1.0
`aa-gamma`	`none`	`Antialias_Gamma`	gamma used when comparing sub-sample colours
Gamma			
`display-gamma`	`none`	`Display_Gamma`	gamma the final image is rendered *for*
`file-gamma`	`none`	`File_Gamma`	gamma assumed for `rgb <...>` colour literals in the scene
Tracing			
`max-trace-level`	`none`	`Max_Trace_Level`	cap on reflection / refraction ray depth
`bounding`	`none`	`Bounding=on/off`	toggle automatic bounding-slab acceleration
`bounding-threshold`	`none`	`Bounding_Threshold`	minimum children before an auto-bound is built
`remove-bounds`	`none`	`Remove_Bounds=on/off`	discard user `bounded_by` when POV-Ray's own bound is tighter
`split-unions`	`none`	`Split_Unions=on/off`	split non-overlapping `union` children into separate bounds
Partial render			
`start-row`	`none`	`Start_Row`	render only this pixel-row range
`end-row`	`none`	`End_Row`	(integers ≥ 1 or fractions 0..1)
`start-column`	`none`	`Start_Column`	pixel-column range
`end-column`	`none`	`End_Column`	
Output encoding			
`output-alpha`	`false`	`Output_Alpha=on/off`	emit RGBA PNG; use with `background { color rgbt <...,1> }`
`compression`	`none`	`Compression (0–9)`	PNG deflate level, `0` none, `9` max
Escape hatch			
`extra`	`()`	`raw command strings appended verbatim`	any flag from POV-Ray's `.ini` / CLI reference

Passing `none` suppresses the flag so POV-Ray's own default stays in force. The plugin always sets `+FN` (PNG), `-D` (no display), and `Work_Threads=1`.
````

6. Resolution & quality

6.1. `width` / `height`

Control the output image size in pixels. Default is 800×600.

```typst
#render("...", width: 1200, height: 900)
#render("...", width: 320, height: 240)
```

The scene's `camera { right (image_width/image_height)*x }` uses these automatically.

6.2. `quality`

Integer 0–11 controlling which rendering features are enabled:

Level	Features enabled
`0`	Ambient light only
`1`	+ diffuse, pigment patterns
`2`	+ point lights, shadows
`5`	+ reflections (no refraction)
`8`	+ refractions, shadow transparency
`9`	+ full radiosity / media sampling **(default)**

Lower quality renders faster — useful for draft iterations. Side-by-side on the same scene at 480×300 (no AA), median of 3 samples on the hardware listed in Section 16:

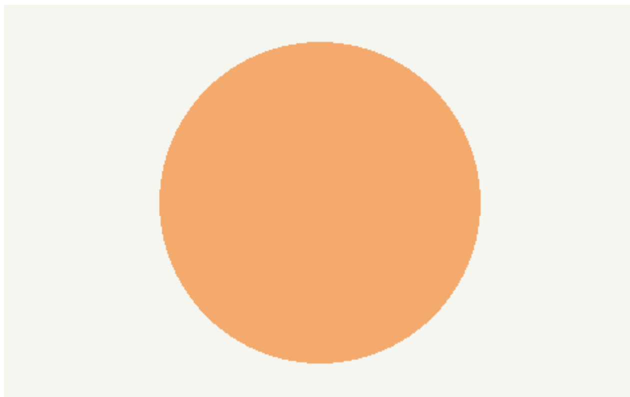


Figure 2: `quality: 0` — ambient only, **0.81 s**. Useful while iterating on layout / framing.

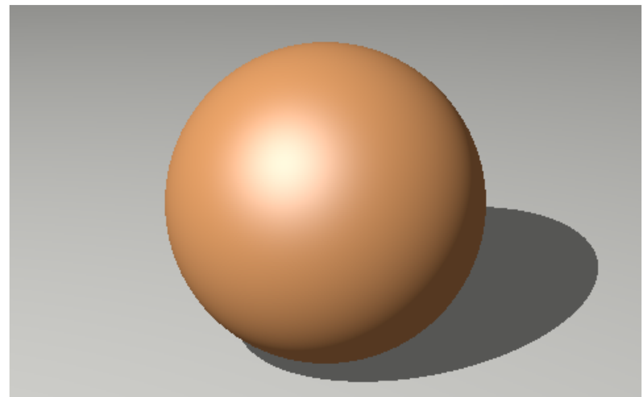


Figure 3: `quality: 9`, full lighting + shadows: **1.60 s**

Lighting-bound scenes benefit most; geometry-bound ones barely move.

7. Antialiasing

Enabled by default. Five kwargs control the behaviour:

kwarg	default	effect
`antialias`	`true`	master switch; `false` skips AA entirely
`aa-threshold`	`0.3`	colour delta that triggers extra samples; lower = smoother/slower
`aa-method`	`2`	1 fixed grid, 2 adaptive recursive, 3 generic oversampling (POV-Ray 3.8+) — not just edge AA, also suppresses moiré, image noise from jittered area lights, subsurface light transport, and micronormals. Parameterisation mirrors adaptive focal blur.
`aa-depth`	`3`	recursion depth; up to depth² samples per pixel
`aa-jitter`	`true`	randomise sub-pixel positions

8. Recipes

Layout scaffolding. When iterating on document layout you don't need photorealism. Drop quality, skip AA, and shrink resolution so each `typst watch` save lands in under a second — restore defaults once layout settles:

```typst
#render("...", quality: 2, antialias: false, width: 320, height: 240)
```

Publication quality. For the camera-ready render, tighten the AA threshold and deepen recursion:

```typst
#render("...", antialias: true, aa-threshold: 0.1, aa-depth: 4,
  width: 1200, height: 900)
```

9. Transparent background

Set `output-alpha: true` and use `background { color rgbt <0,0,0,1> }` in the scene (`t = 1` means fully transparent). The result is an RGBA PNG that composites over the Typst page background.

```typst
#pov("background { color rgbt <0, 0, 0, 1> }
      camera { location <0, 1, -4> look_at 0 }
      light_source { <4, 6, -5> rgb 1.3 }
      sphere { 0, 1 pigment { rgb <0.2, 0.55, 0.9> } }",
  output-alpha: true)
```

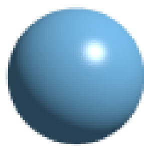


Figure 4: On white page

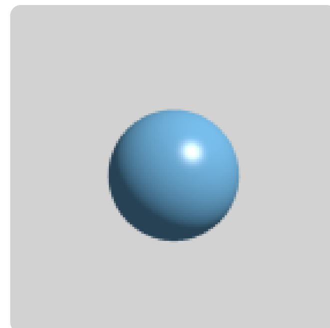


Figure 5: On grey — alpha at work

Without `output-alpha: true`, the background is opaque black regardless of `rgbt`.

10. Multi-file scenes (`includes`)

Split a POV-Ray project across `.inc` files by passing the `includes` dict — `render()` textually expands every `#include "name"` before handing the scene to the plugin. Nested includes work; a cycle guard prevents `a.inc → b.inc → a.inc` from looping.

```typst
#render(read("scene.pov"),
  includes: ("materials.inc": read("materials.inc")))
```

11. Typst-driven scene parameterisation

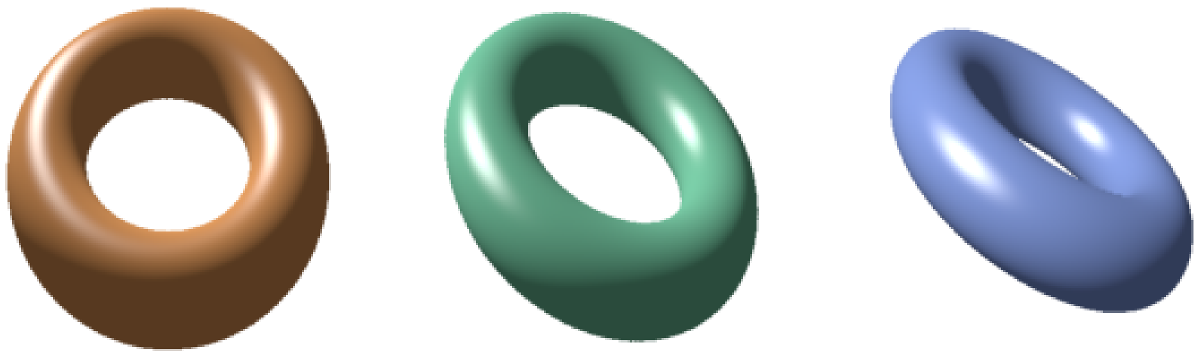
The first argument to `render()` is just a string — build it from any Typst values (loop index, palette entry, computed angle) to drive geometry, colour, or camera. The grid below is produced by a single `map` over three angles:

typst

```
#let palette = (rgb("#f26924"), rgb("#3ab573"), rgb("#4c6ef5"))
#let pov-rgb(c) = {
  let (r, g, b, ..) = c.components()
  "<" + str(r/100%) + ", " + str(g/100%) + ", " + str(b/100%) + ">"
}

#let orbit(theta, col) = (
  "global_settings { assumed_gamma 1.0 max_trace_level 2 }\n"
  + "background { color rgbt <0, 0, 0, 1> }\n"
  + "camera { location <" + str(5 * calc.cos(theta))
  + ", 2, " + str(5 * calc.sin(theta)) + "> look_at 0 angle 35 }\n"
  + "light_source { <4, 6, -4> rgb 1.3 }\n"
  + "torus { 1, 0.32 pigment { rgb " + pov-rgb(col) + " } "
  + "finish { specular 0.8 roughness 0.02 } rotate x*25 no_shadow }"
)

#grid(
  columns: (1fr, 1fr, 1fr), column-gutter: 12pt,
  ..((-90deg, 0), (-45deg, 1), (0deg, 2)).map(((a, i)) => render(
    orbit(a, palette.at(i)),
    width: 180, height: 180, quality: 2,
    antialias: true, aa-threshold: 0.7, aa-depth: 2, output-alpha: true,
  )),
)
```



12. Tracing depth & advanced options

12.1. `max-trace-level`

Caps reflection / refraction ray bounces (POV-Ray default 5, overrides any in-scene `global_settings`). Drop to 2 for diffuse scenes; raise for glass-on-glass or hall-of-mirrors.

typst

```
#render("...", max-trace-level: 10)
```

12.2. `extra` (escape hatch)

Array of raw POV-Ray command strings appended verbatim — any flag from POV-Ray's `.ini` / CLI reference without a typed kwarg.

```typst
#render("...", extra: ("+WV3.8", "+HI"))
```

13. Examples

13.1. Constructive solid geometry

POV-Ray's signature feature: combine primitives via the boolean operators `union`, `intersection`, `difference`, and `merge`. Each takes any number of objects (including other CSG objects) and produces a new object that participates in further CSG. This is what lets a few dozen lines of SDL describe shapes that would take a mesh modeller hours.

The example below is built in three nested operations: `intersection { box ∩ sphere }` rounds the cube's corners, `union { 3 cylinders }` collects three orthogonal rods, and `difference { rounded-cube − rods }` carves the rods out, exposing the rounded interior surfaces.

csg.pov

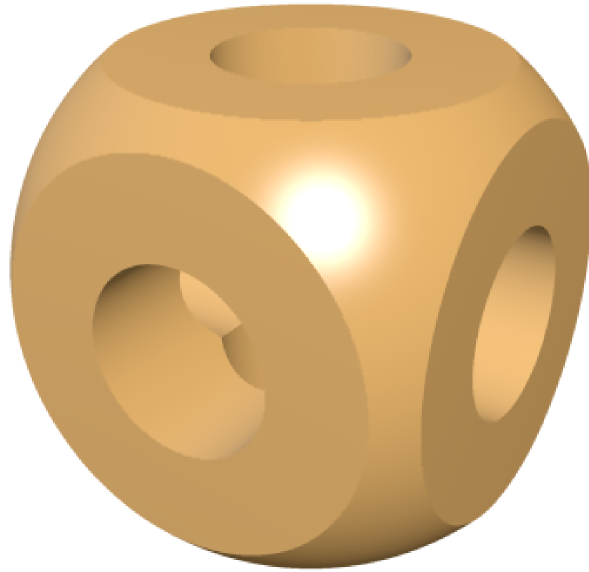
```povray
global_settings { assumed_gamma 1.0 max_trace_level 2 }
background { color rgbt <0, 0, 0, 1> }
camera { location <2.6, 2.4, -4.2> look_at <0, 0, 0> angle 38 }
light_source { < 6, 9, -8> color rgb 1.4 }
light_source { <-4, 2, -2> color rgb <0.4, 0.5, 0.95> shadowless }

#declare RoundedCube = intersection {
  box { -<1,1,1>, <1,1,1> }
  sphere { 0, 1.32 }
}

#declare Bores = union {
  cylinder { -1.2*x, 1.2*x, 0.42 }
  cylinder { -1.2*y, 1.2*y, 0.42 }
  cylinder { -1.2*z, 1.2*z, 0.42 }
}

difference {
  object { RoundedCube }
  object { Bores }
  pigment { rgb <0.96, 0.55, 0.18> }
  finish { ambient 0.12 diffuse 0.55 specular 1.0 roughness 0.01 }
  no_shadow
}
```

```typst
#render(read("csg.pov"), output-alpha: true)
```



Rounded cube with three orthogonal cylindrical bores — `intersection`, `union`, and `difference` in one object.

13.2. Quaternion Julia fractal

Iterate the map

$$z_{n+1} = z_n^2 + c, \quad z \in \mathbb{H}, \quad c = -0.083 - 0.83j - 0.025k$$

over the quaternions $\mathbb{H}$. The rendered set is the boundary between starting points z_0 whose orbit stays bounded and those that escape to infinity (bailout at $n = 8$). A 3D slice of the 4-dimensional Julia set. See Wikipedia on quaternion Julia sets.

julia.pov

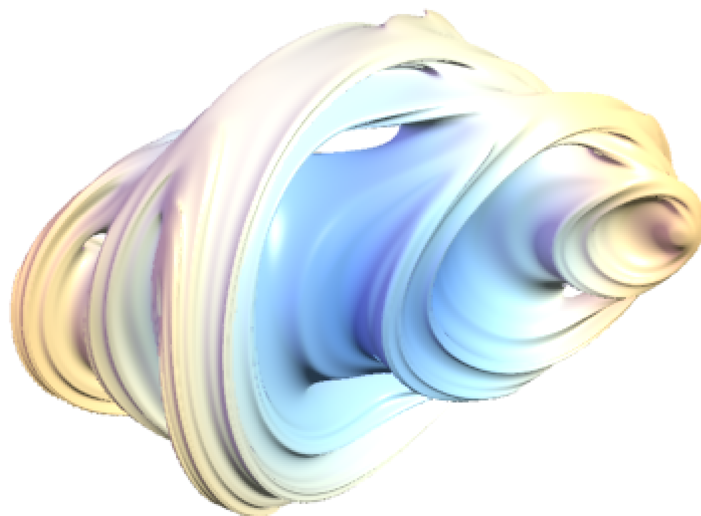
```povray
global_settings { assumed_gamma 1.0 max_trace_level 2 }
background { color rgbt <0, 0, 0, 1> }

camera { location <0, 2, -4.5> look_at <0, 0, 0> angle 38 }
light_source { <4, 6, -6> color rgb 1.6 }
light_source { <-5, 3, -2> color rgb <0.5, 0.3, 0.8> shadowless }

julia_fractal {
  <-0.083, 0.0, -0.83, -0.025>
  quaternion
  sqr
  max_iteration 8
  precision 50
  slice <0, 0, 0, 1>, 0.15

  pigment {
    spherical
    color_map {
      [0.00 rgb <0.95, 0.55, 0.18>]
      [0.30 rgb <0.95, 0.78, 0.40>]
      [0.55 rgb <0.45, 0.80, 0.95>]
      [0.80 rgb <0.10, 0.30, 0.80>]
      [1.00 rgb <0.05, 0.06, 0.25>]
    }
    scale 1.6
  }
  finish {
    ambient 0.08
    diffuse 0.85
    specular 0.4
    roughness 0.020
  }
  rotate <-20, 30, 0>
  no_shadow
}
```

```typst
#render(read("julia.pov"), antialias: true, aa-threshold: 0.4, aa-depth: 2, output-alpha: true)
```



Quaternion Julia set for $c = -0.083 - 0.83j - 0.025k$, 3D slice at $q_w = 0.15$, coloured radially from origin.

13.3. Clebsch diagonal cubic

In Cartesian coordinates $(x, y, z) \in \mathbb{R}^3$, the Clebsch surface is the zero set of the symmetric cubic polynomial

$$81(x^3 + y^3 + z^3) - 189(x^2y + x^2z + xy^2 + y^2z + xz^2 + yz^2)$$
$$+54xyz + 126(xy + yz + zx) - 9(x^2 + y^2 + z^2) - 9(x + y + z) + 1 = 0$$

A smooth cubic surface with full tetrahedral symmetry containing exactly 27 real straight lines (Cayley–Salmon, 1849; Clebsch's explicit coordinates, 1869). POV-Ray's `cubic { ... }` takes the 20 polynomial coefficients directly and ray-marches the zero set via Sturm sequences — no mesh, no approximation. See Wikipedia on the Clebsch surface.

clebsch.pov

```povray
global_settings { assumed_gamma 1.0 max_trace_level 2 }
background { color rgbt <0, 0, 0, 1> }

camera { location <1.7, 1.5, -5.4> look_at <0, 0, 0> angle 42 }

light_source { <-6, 10, -10> color rgb 1.6 }
light_source { < 8,  2,  -2> color rgb <0.20, 0.30, 0.60> shadowless }

cubic {
  <  81, -189, -189, -9, -189, 54, 126, -189, 126, -9,
     81, -189, -9, -189, 126, -9, 81, -9, -9, 1 >
  bounded_by { sphere { 0, 2 } }
  clipped_by { sphere { 0, 1.6 } }
  pigment {
    gradient y
    color_map {
      [0.00 rgb <0.05, 0.10, 0.45>]
      [0.40 rgb <0.10, 0.40, 0.92>]
      [0.75 rgb <0.45, 0.80, 0.98>]
      [1.00 rgb <0.85, 0.95, 1.00>]
    }
    scale 3 translate <0, -1.5, 0>
  }
  finish { ambient 0.07 diffuse 0.85 specular 0.5 roughness 0.025 }
  no_shadow
}
```

```typst
#render(read("clebsch.pov"), output-alpha: true)
```

Clebsch diagonal cubic, clipped to a sphere; stepped vertical gradient pigment, the contour bands tracing the surface's height curvature.

13.4. Schoen gyroid

In Cartesian coordinates $(x, y, z) \in \mathbb{R}^3$, Schoen's gyroid is a **triply periodic minimal surface** — the zero set of the transcendental function

$$f(x, y, z) = \sin x \cos y + \sin y \cos z + \sin z \cos x = 0$$

It locally minimises area (mean curvature zero everywhere) and tiles $\mathbb{R}^3$ with cubic periodicity. Discovered by Alan Schoen at NASA in 1970. POV-Ray renders it via an `isosurface`.

gyroid.pov

```povray
global_settings { assumed_gamma 1.0 max_trace_level 2 }
background { color rgbt <0, 0, 0, 1> }

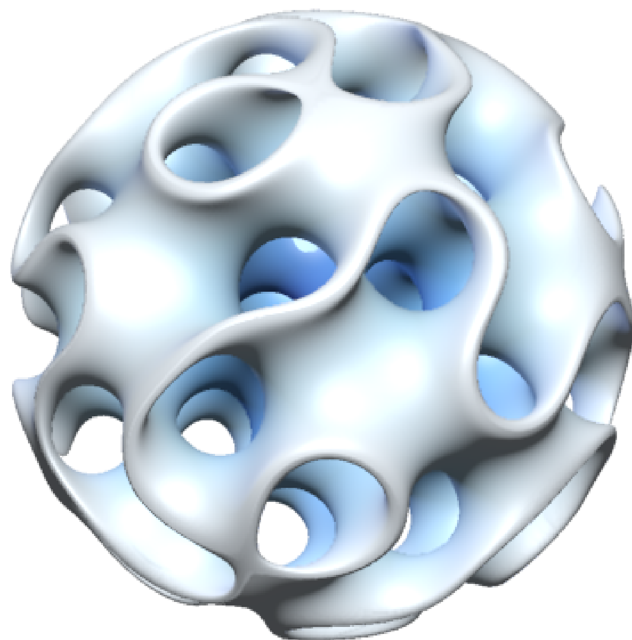
camera { location <8, 6, -11> look_at <0, -0.3, 0> angle 44 }

light_source { <6, 10, -8> color rgb 1.2 }
light_source { <-5, 1, -2> color rgb <0.22, 0.28, 0.50> shadowless }

#declare K = 2.0;
#declare R = 4;
#declare Slab     = function(x,y,z) {
  abs(sin(K*x)*cos(K*y) + sin(K*y)*cos(K*z) + sin(K*z)*cos(K*x))
  - (0.15 + 0.32 * (x*x + y*y + z*z) / (R*R))
}
#declare SphereSD = function(x,y,z) { sqrt(x*x + y*y + z*z) - R }
#declare SmoothMax = function(a, b, k) {
  (a + b + sqrt((a - b)*(a - b) + k*k)) / 2
}

isosurface {
  function { SmoothMax(Slab(x, y, z), SphereSD(x, y, z), 0.25) }
  contained_by { sphere { 0, R + 0.15 } }
  threshold 0
  accuracy 0.018
  max_gradient 2
  open
  pigment {
    spherical
    color_map {
      [0.00 rgb <0.85, 0.90, 0.95>]
      [0.20 rgb <0.55, 0.80, 0.95>]
      [0.50 rgb <0.20, 0.50, 0.92>]
      [0.80 rgb <0.10, 0.25, 0.72>]
      [1.00 rgb <0.15, 0.08, 0.40>]
    }
    scale R
  }
  finish { ambient 0.06 diffuse 0.88 specular 0.3 roughness 0.04 }
  no_shadow
}
```

```typst
#render(read("gyroid.pov"), antialias: true, aa-threshold: 0.7, aa-depth: 2, output-alpha: true)
```



Schoen gyroid as a thin shell around the zero set; clipped to a sphere.

13.5. Lissajous knot 8_{21}

Parametric space curve

$$\boldsymbol{r}(t) = 0.75 \begin{pmatrix} \cos(3t + 0.1) \\ \cos(4t + 0.7) \\ \cos(7t) \end{pmatrix}, \quad t \in [0, 2\pi]$$

with frequencies $(3, 4, 7)$ (pairwise coprime, phases $(0.1, 0.7, 0)$) — a prime knot with 8 crossings, classified as 8_{21} in the Rolfsen table. See Wikipedia on Lissajous knots.

knot.pov

```povray
background { color rgbt <0, 0, 0, 1> }
global_settings { assumed_gamma 1.0
max_trace_level 2 }

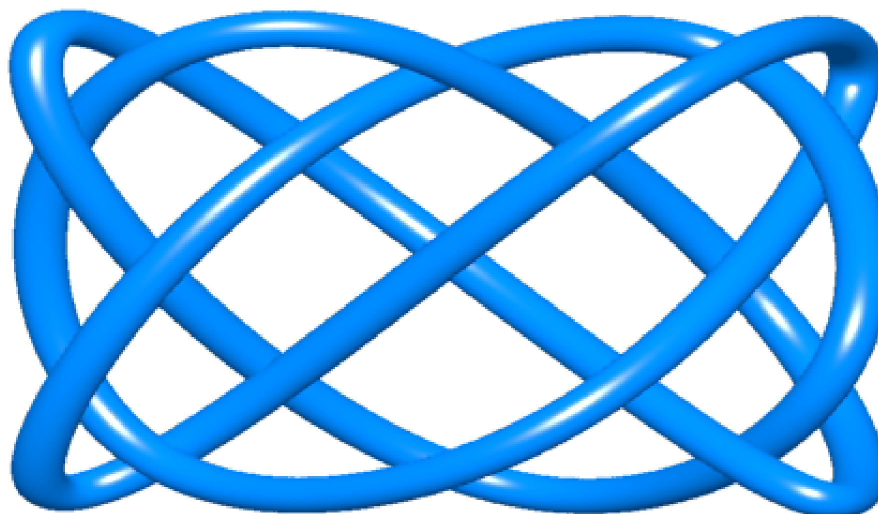
camera
  {
  location <0, 0, -25>
  right <1,0,0> up <0,1,0>
  look_at  <0, 0, 0>
  angle 3.5
  }

light_source { <0, 20, -50> color rgb 1.6 }

#declare r_tube = 0.04;
#declare num_steps = 17;
#declare step_size = 1/num_steps;

sphere_sweep
  {
  b_spline num_steps+3,
  #declare N = -1;
  #while(N <= num_steps + 1)
    #declare theta = 2 * pi * N * step_size;
    < 0.75*cos(3*theta + 0.1),
      0.75*cos(4*theta + 0.7),
      0.75*cos(7*theta) >, r_tube
    #declare N = N + 1;
  #end
  pigment { color rgb <0, 0.25, 1> }
  finish
    {
    ambient 0.18
    diffuse 0.75
    specular 0.95
    roughness 0.012
    }
  no_shadow
  }
```

```typst
#render(read("knot.pov"), antialias: true, aa-threshold: 0.7, aa-depth: 2, output-alpha: true)
```



Lissajous knot 8_{21}

13.6. Hopf fibration

In geometry, the Hopf fibration gives a partition of the 3-dimensional sphere S^3 by great circles. More precisely, it defines a fibred structure on S^3: the base space is the 2-dimensional sphere S^2, and the model fibre is a circle S^1. In particular, this means there exists a projection map $p : S^3 \to S^2$ whose preimages of each point of S^2 are circles. This structure was discovered by Heinz Hopf in 1931. See Wikipedia on the Hopf fibration.

hopf.pov

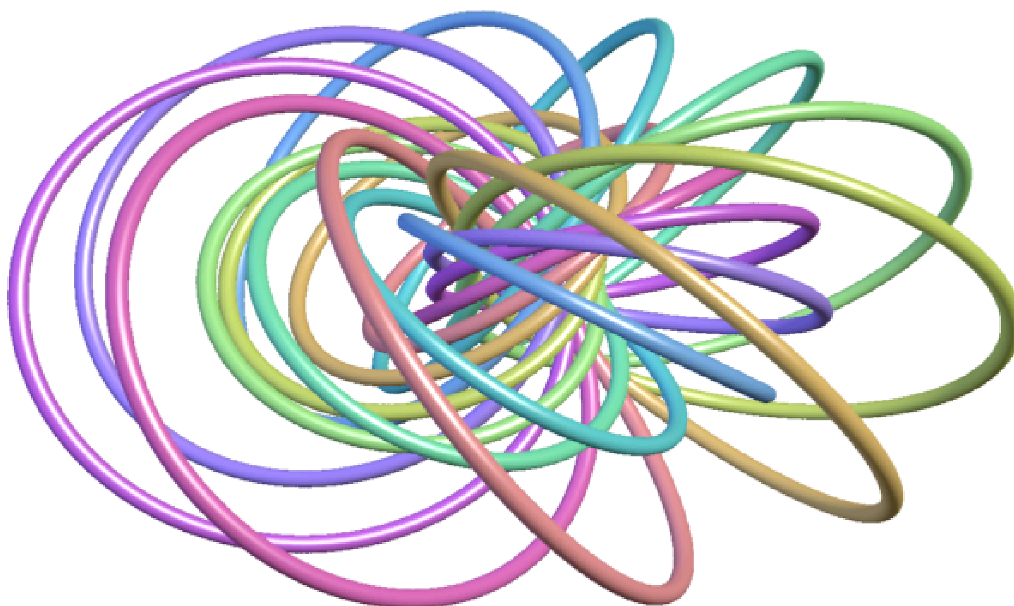
```povray
global_settings { assumed_gamma 1.0 max_trace_level 2 }
background { color rgbt <0, 0, 0, 1> }
camera { location <0, 4.6, -10.5> look_at <0, -0.35, 0> angle 32 }
light_source { < 8, 12, -10> color rgb 1.4 }
light_source { <-6,  2,  -3> color rgb <0.35, 0.45, 0.95> shadowless }

#declare r_tube = 0.06;

#macro HopfFiber(theta, phi, hue)
  #local st = sin(theta/2);
  #local ct = cos(theta/2);
  #macro F(psi)
    <ct*cos(psi), st*cos(psi+phi), ct*sin(psi)> / (1 - st*sin(psi+phi))
  #end
  #local P0 = F(0); #local P1 = F(2*pi/3); #local P2 = F(4*pi/3);
  #local n  = vcross(P1-P0, P2-P0);
  #local a2 = vdot(P2-P1, P2-P1);
  #local b2 = vdot(P0-P2, P0-P2);
  #local c2 = vdot(P1-P0, P1-P0);
  #local Cn = (a2*(b2+c2-a2)*P0 + b2*(c2+a2-b2)*P1 + c2*(a2+b2-c2)*P2) / (4*vdot(n,n));
  #local N  = vnormalize(n);
  #local T  = (abs(N.y) < 0.9 ? <0,1,0> : <1,0,0>);
  #local E1 = vnormalize(vcross(T, N));
  #local E3 = vcross(N, E1);
  torus {
    vlength(P0-Cn), r_tube
    pigment { rgb <0.55+0.4*cos(2*pi*hue), 0.55+0.4*cos(2*pi*hue+2*pi/3), 0.55+0.4*cos(2*pi*hue+4*pi/3)> }
    finish { ambient 0.1 diffuse 0.55 specular 0.8 roughness 0.02 }
    matrix <E1.x,E1.y,E1.z, N.x,N.y,N.z, E3.x,E3.y,E3.z, Cn.x,Cn.y,Cn.z>
    no_shadow
  }
#end

#declare T = array[2] { 2*pi/5, 3*pi/5 };
#local i = 0;
#while (i < 2)
  #local j = 0;
  #while (j < 10)
    HopfFiber(T[i], 2*pi*j/10, j/10 + i/2)
    #local j = j + 1;
  #end
  #local i = i + 1;
#end
```

```typst
#render(read("hopf.pov"), antialias: true, aa-threshold: 0.7, aa-depth: 2, output-alpha: true)
```



20 Hopf fibres over two parallels of S^2, coloured by hue of base point.

13.7. Advanced lighting

POV-Ray ships advanced light-transport features — refraction with chromatic dispersion, photon-mapped caustics, radiosity for indirect illumination, area lights with soft shadows, and volumetric media. This example exercises one of them: a wavy water surface (index of refraction $n = 1.33$) refracts incident sunlight by Snell's law, and the bent rays concentrate onto the pool floor as the iconic shimmering caustic mesh. POV-Ray computes this by Jensen's photon mapping (1996): a pre-pass shoots photons from the light, tracks each through the wave-perturbed water normals, and stores landing positions in a kd-tree; the render then gathers nearby photons at each surface point. See Wikipedia on caustics and photon mapping.

caustic.pov

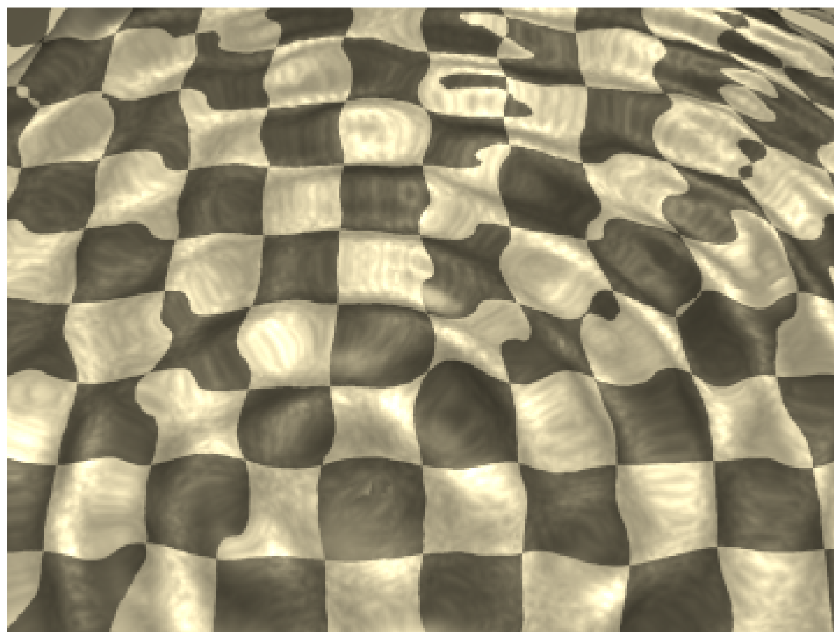
```povray
global_settings {
  assumed_gamma 1.0
  max_trace_level 3
  photons { spacing 0.10 gather 3, 8 }
}
background { color rgbt <0, 0, 0, 1> }
camera { location <0, 5.5, -3.5> look_at <0, -1.6,
1.2> angle 38 }

light_source {
  <-2, 15, -4> color rgb <1.0, 0.88, 0.55> * 2.4
  photons { refraction on reflection off }
}

plane {
  y, -2
  pigment {
    checker
    color rgb <0.05, 0.05, 0.05>,
    color rgb <0.29, 0.29, 0.29>
    scale 0.6
  }
  finish { ambient 0.04 diffuse 0.85 }
}

disc {
  <0, 0, 0>, y, 4
  pigment { rgbt <0.85, 0.92, 1.0, 0.94> }
  finish {
    ambient 0
    diffuse 0.02
    specular 0.3
    roughness 0.012
    reflection 0.04
  }
  normal {
    ripples 0.6
    frequency 2.5
    turbulence 0.15
    scale 1.2
  }
  interior { ior 1.33 }
  photons { target refraction on collect off }
}
```

```typst
#render(read("caustic.pov"))
```



Refractive caustics under a wavy water surface ($n = 1.33$); photon mapping concentrates the sunlight onto the pool floor.

14. Debugging

POV-Ray parse-time errors surface as typst compile errors with a gcc-style location and source-line preview, courtesy of a parser hook in our build. `#error "msg"` is the supported authoring-time abort — test it from your scene to verify the diagnostic plumbing.

For visual orientation, draw a coordinate frame alongside the scene:

debug-axes.pov

```povray
global_settings { assumed_gamma 1.0 max_trace_level 2 }
background { color rgbt <0, 0, 0, 1> }

camera {
  location <25, 25, 25>
  look_at  <0, 0.7, 0>
  angle 8
}

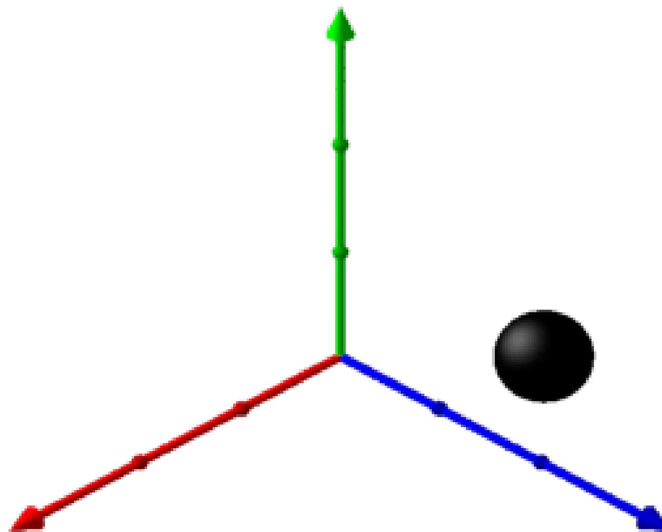
light_source { <4, 6, -4> rgb 1.4 }

sphere {
  <0, 1, 2>, 0.35
  finish { specular 0.1 diffuse 0.1 }
}

#macro Axis(dir, col, len)
  cylinder { 0, len*dir, 0.035
    pigment { rgb col } finish { ambient 0.2 diffuse 0.7 } no_shadow }
  cone { len*dir, 0.10, (len+0.25)*dir, 0
    pigment { rgb col } finish { ambient 0.2 diffuse 0.7 } no_shadow }
  #local i = 1;
  #while (i <= len)
    sphere { i*dir, 0.06
      pigment { rgb col*0.6 } finish { ambient 0.2 diffuse 0.7 } no_shadow }
    #local i = i + 1;
  #end
#end

Axis(x, <1.0, 0.0, 0.0>, 3)
Axis(y, <0.0, 1.0, 0.0>, 3)
Axis(z, <0.0, 0.0, 1.0>, 3)
```

```typst
#render(read("debug-axes.pov"), antialias: true, aa-threshold: 0.7, aa-depth: 2, output-alpha: true)
```



Origin-centred RGB axes (red = x, green = y, blue = z)

15. Further reading and other examples

- **POV-Ray official documentation**
  https://www.povray.org/documentation/
- **Paul Bourke — Geometry**
  https://paulbourke.net/geometry/
  Marching cubes, minimal surfaces, Platonic solids, aperiodic tilings, most with POV-Ray source.
- **POV-Ray official — Object and Scene Files**
  https://www.povray.org/resources/links/POV-Ray_Include_Macro_and_Object_Files/Object_and_Scene_Files/
  Community scene files and reusable macro libraries.
- **Friedrich Lohmüller — POV-Ray tutorial on shapes**
  https://www.f-lohmueller.de/pov_tut/all_shapes/shapes000e.htm
  Visual walkthrough of every primitive and CSG idiom, from spheres and cylinders through text, isosurfaces, and height fields — each with its SDL snippet and rendered output.
- **Michael Scharrer — POV-Ray scenes**
  https://mscharrer.net/povray/scenes/
  22 algorithmic and fractal scenes, strong on CSG composition and mathematical visualisation; source available per scene.
- http://dataduppedings.no/subcube/POV-Ray/index.html, http://bugman123.com/, …

16. Render times

Render times for each gallery scene, measured with an AMD Ryzen 9 9950X (Zen 5, 5.7 GHz).

Scene	Resolution	AA (threshold / depth)	Median time
`basic.pov`	480×300	—	2.3 s
`csg.pov`	480×360	—	3.4 s
`julia.pov`	420×315	0.4 / 2	4.3 s
`clebsch.pov`	480×360	—	4.1 s
`gyroid.pov`	520×390	0.7 / 2	19.3 s
`knot.pov`	480×320	0.7 / 2	15.1 s
`hopf.pov`	600×400	0.7 / 2	7.7 s
`caustic.pov`	360×270	—	13.5 s

The Gyroid takes longer because I tweaked it to look stylized/beautiful (clipped smoothly with a bounding sphere), and unfortunately raymarching isn't free. Refraction neither, same for caustic.pov.

17. Conclusion

POV-Ray has a lot of features, too many to show all of them here. Feel free to read its documentation to learn about its possibilities and look for examples online. Many wikipedia pages (in algebraic geometry, topology, and so on) use POV-Ray to visualize all sorts of things: https://commons.wikimedia.org/wiki/Category:POV-Ray