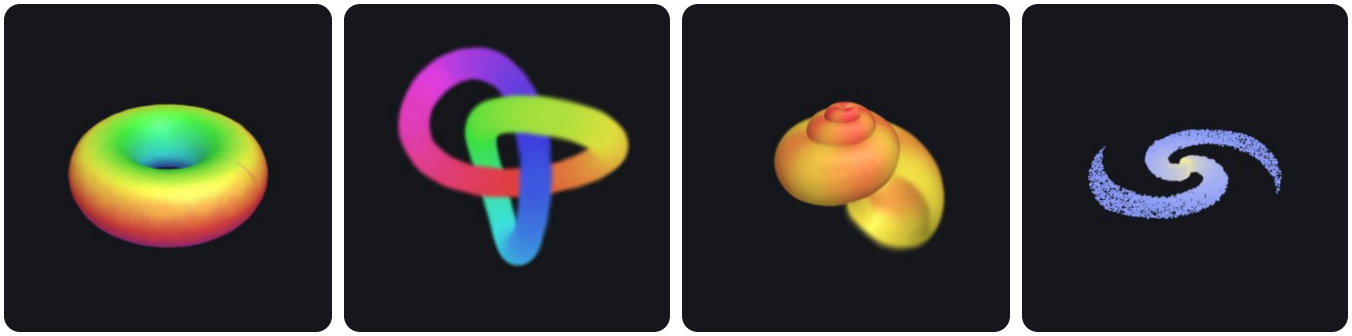
SPLASHERY · UPDATED OCTOBER 11, 2026

The Tinkerer's Manual

How to program Gaussian splats: what one splat is, how the toys are built, and a small language for writing your own splat equations, and toys that show pictures and read what you type.



Almost every toy in the Splashery Playground is made of Gaussian splats: tens of thousands of soft, colored blobs that blend into a solid-looking thing. This manual is for anyone who knows some math and wants to make splats do what they say. It goes chapter by chapter. You can stop after any chapter and still have something to play with.

Splashery is primarily splat based, but not entirely. Two honest exceptions:

- The photo toys (Photo to 3D and Moving photo to 3D) open in a **Sharp view**, a second way to see the same photo: the picture at its full resolution on a textured relief (a mesh of triangles) lifted by the same depth, so small text stays readable. That view is not drawn with splats; the splats are switched off while it shows. The Splats / Sharp switch in the toy's panel brings the splats back.
- The liquids and flames are part splats and part something else: see "What else Splashery does now" for how each fluid is drawn. (The Studio toys are behind the labs switch; see the same chapter.)

Contents

Chapter 1: What a splat is p. 5

- The numbers in one splat p. 7
- The Gaussian: a bell in three dimensions p. 11
- The covariance, built from sizes and a rotation p. 12
- From 3D to the screen p. 14

- Sorting and blending, back to front p. 15

Chapter 2: How a toy is built p. 20

- Two places a toy lives p. 20
- Shapes from the kit p. 21
- How many splats? What a bigger budget does p. 21
- Colors from functions p. 25
- Parts that hinge, spin or slide p. 25
- Behaviors p. 27
- A small, complete recipe p. 27
- Run it yourself p. 30

Chapter 3: The splat equation language p. 32

- The idea: every splat gets its own u and v p. 32
- The fields p. 33
- Expressions p. 35
- Time p. 38
- What the toy does for you p. 42
- Why it is safe p. 43

Chapter 4: A gallery of programs p. 44

- Sixty more programs p. 51

Chapter 5: Pictures, screens, and your own input p. 67

- Picture sheets: a page, a photo, a GIF, a video p. 67
- Turning a page p. 69
- Your own input: typing, drawing, opening a file p. 71
- The Screen and the Gaussian splatting toy p. 73
- Steering the view: the top bar and the tilt lock p. 75

Chapter 6: Moving things p. 76

- Hands-on play: pick up, toss, settle p. 76
- Pieces and joints: parts that move the way the real thing's do p. 78
- Ropes, cloth, and soft stretch p. 81
- Materials: a thrown thing moves like the real thing p. 82
- Rigid parts and the motion driver p. 82
- Where to try it p. 83

Chapter 7: Real data as splats p. 84

- Molecules p. 84
- Earthquakes p. 85
- The night sky p. 86
- Volumes and point clouds p. 88

- Weather, maps, and the other data toys p. 89
- The rules for live data p. 91
- Where to try it p. 91

Chapter 8: Photos and video into 3D p. 92

- Photo to 3D p. 92
- The Sharp view p. 93
- Moving photo to 3D p. 94
- Video to 3D p. 95
- The photoreal captures p. 96
- Where to try it p. 97

Chapter 9: Codes and pages p. 98

- QR codes p. 98
- More QR toys, and other barcodes p. 100
- Pages, PDFs, and GIFs p. 101
- Share links p. 103
- Where to try it p. 103

Chapter 10: Make your own p. 104

- Five ways to make your own p. 104
- Roll one and open one p. 104
- The Studio p. 105
- Models, and your splat files p. 106
- Writing a recipe p. 108
- Sharing what you made p. 109
- Where to try it p. 109

What else Splashery does now p. 110

- Hands-on physics p. 110
- The QR code toy p. 110
- Captures and file formats p. 112
- Camera, microphone, and screen p. 113
- Books, albums, and frames p. 113
- Studio: sound and photos p. 114
- Models to splats p. 114
- Worlds p. 115
- Fluids, songs, and smooth bodies p. 116
- Solid, Fine, and Dots, and the detail you see p. 118
- The viewer's controls p. 118

What Splashery can do that other splat tools can't p. 120

How Splashery is built p. 123
- Plain files, and nothing else p. 126
- Where the files are p. 127
- What PlayCanvas does, and what is our own p. 128
- Every library that ships, and when it loads p. 129
- How a toy gets on the shelf p. 132

Share links and QR codes p. 133
- What is in a link p. 133
- Why one changed character usually breaks it p. 134
- Old links keep loading p. 135
- How long a link can be p. 135
- A scene as a QR code p. 135

Open questions: other choices, not tried yet p. 137

What comes later: Code a toy p. 138

Glossary p. 139

Further reading p. 146
- Where splats come from p. 146
- Robotics and other uses p. 146
- Moving, talking, and generated splats p. 147
- Open projects and tools p. 148

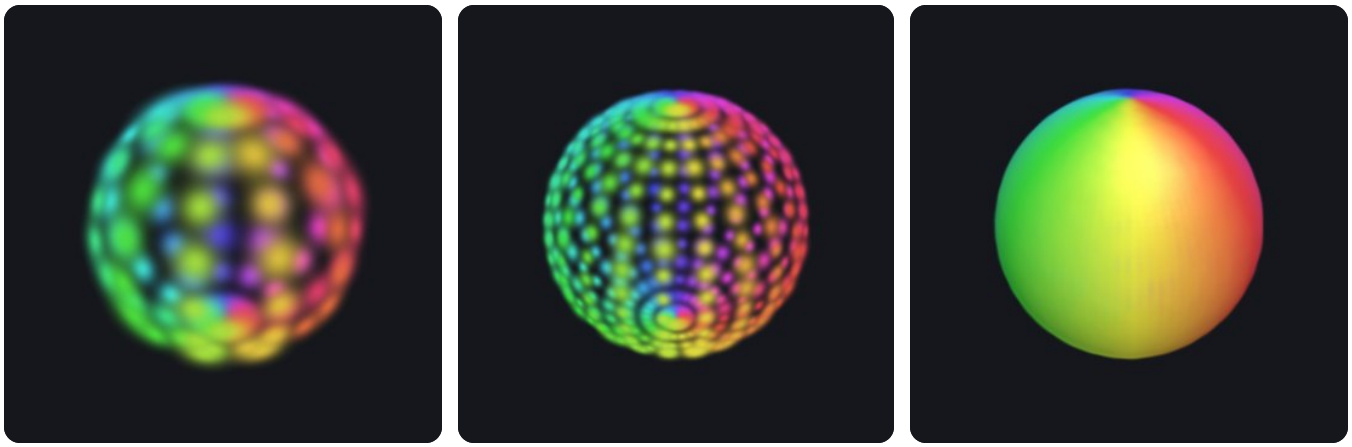
References p. 149

Quick reference p. 151
- One splat p. 151
- A splat program p. 151
- A picture toy p. 151
- A recipe (a toy) p. 151
- The controls p. 152
- Key numbers p. 152
- What goes where p. 153

CHAPTER 1

What a splat is

A **splat** is a small cloud of color floating in space. Up close, it looks like a soft ellipse, bright in the middle and fading at the edges. On its own, it is nothing much. Put thousands side by side, however, and they melt into surfaces: a torus, a seashell, a cat – practically any object imaginable can be built from splats!



100 splats | 400 splats | 6,000 splats

The same sphere with more and more splats, made with the Splat equation toy. With few splats you see each soft blob; with enough of them, they blend into one smooth surface.

What is a 3D Gaussian?

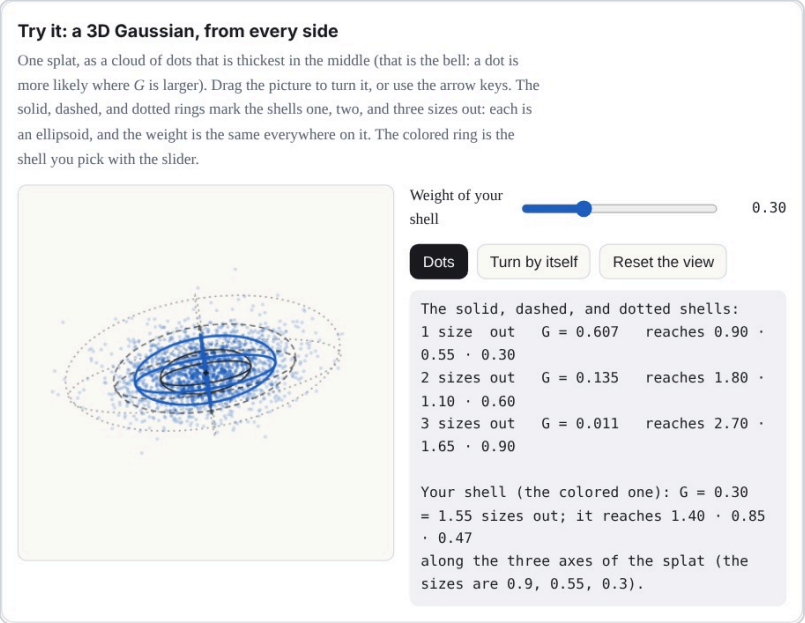
Think of a smooth weight field centered at a point μ. Three positive widths and a rotation shape it: the places where the weight is equal form nested ellipsoids. The weight G is 1 at the center and falls off with distance measured in units of those widths. To draw one, the renderer flattens the field into an elliptical patch on the screen, gives it a color, and blends its opacity over the patches behind it. A splat is neither a surface of triangles nor a rigid little ball. Here G peaks at 1, so it is a weight, not a probability density (which would be scaled to add up to 1).

Two terms: weight field and positive widths

A **weight field** is a number at every point in space: 1 at the center, and falling off the farther you go. The three **positive widths** are the splat's sizes along its own three axes. They must be above zero, because a width of zero would make the formula below divide by zero.

Three ways splats come to exist (Figure 1):

- **A recipe's placed splats.** A toy's recipe places them by rule: points sampled on the shapes of **the kit** (Splashery's toy-building library, Chapter 2), each given a size, a rotation, and a color. Nothing is fitted to photos.
- **A trained capture.** A splat file made by other software from many photos of a real thing, then opened here (`.ply` , `.sog` , `.splat` , or `.spz`). The page shows these; it does not train them.
- **The one-photo teaching fit.** The Gaussian splatting toy fits flat, 2D Gaussians to a single picture by gradient descent. It shows the fitting idea; it is not a 3D reconstruction.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/splat.html#demo-gaussian

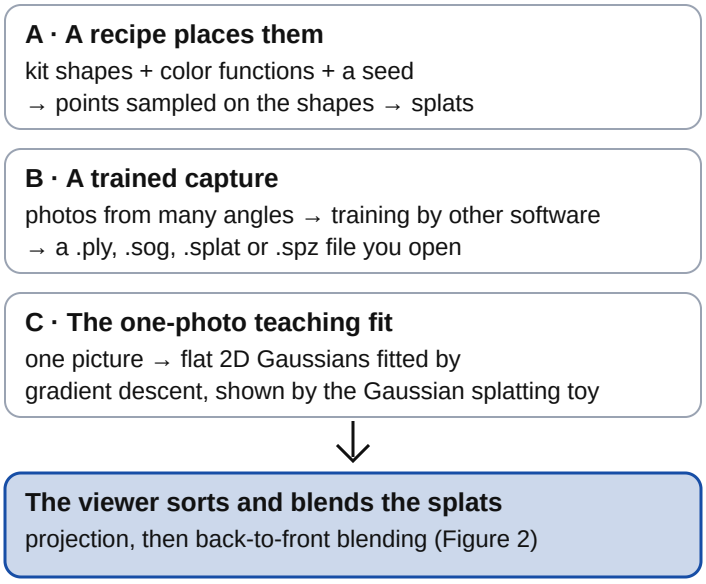


Figure 1. Three different things are called "splats". Only the first two make a 3D object; the third fits a flat picture.

The numbers in one splat

Each splat is described by a handful of numbers:

Part	Numbers	What it means
Center μ	3: x, y, z	Where the middle of the splat is.
Sizes s	3: s_1, s_2, s_3	How far it spreads along each of its own three axes. Equal sizes make a ball; one small size makes a flat disc.
Rotation q	4 (a quaternion)	Which way those three axes point.
Color	3: red, green, blue	Each from 0 to 1.
Opacity α	1	How solid its middle is, from 0 (clear) to 1 (solid).



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/splat.html#demo-numbers

That is 14 numbers per splat. A kit toy starts with a budget of 60,000 splats at the Low tier, 140,000 at Mid, 200,000 at High, and 280,000 at Max. Recipes can raise that budget, up to 400,000 at Max. Auto picks Low, Mid, or High; Max is a viewer choice. Many splat files also let a splat's color change a little with the direction you look from; the toys you build here use one color per splat.

Why these numbers? The splat budgets, and what Auto does

OUR CHOICE Nothing in the splat file format or the browser sets these four numbers.

Each tier is a splat count that a device of that class can draw smoothly. **Auto** reads the device to pick one: Low for 2 GB of memory or less, or two cores or fewer; Mid for a touch-screen device whose shorter side is under 820 pixels, or 4 GB of memory or less, or four cores or fewer; High for the rest. It never picks Max. That is a choice you make in Detail, along with High. Auto also steps down while you drag or paint (an animation alone does not trigger it, by default since September 29, 2026): when frames take longer than about 24 milliseconds, it first lowers the screen resolution, and if frames at the lowest resolution still average over 40 milliseconds for 30 frames in a row, it drops one tier. The project's notes call the tier thresholds and the 24 and 40 millisecond limits "guesses, not tuned on real phones".

What the measurements say. The Sharpness lane (September 29, 2026) measured 14 toys on a 390 by 844 phone in a software renderer. A budget of 200,000 splats instead of 140,000 made no toy sharper, and several shimmered more (the penguin's shimmer went from 2.72 to 3.90), because each splat got smaller. Lifting the cap on the screen's pixel ratio was the real lever: edges came out about a third narrower, and speckle fell by 10 to 60 percent. More splats help a toy's shape, not its grain.

Show the table: what a bigger budget did

Toy (home view)	Mid base budget 140,000 splats, 2x phone (September 29, 2026)	High base budget 200,000 splats, 2x phone	3x phone, pixel-ratio cap lifted
Lamp	0.75 · 0.42 · 1.95	0.72 · 0.45 · 2.03	0.52 · 0.37 · 1.57
Penguin	0.72 · 0.53 · 2.72	0.73 · 0.56 · **3.90**	0.50 · 0.40 · 2.08
Klein bottle	0.84 · 1.19 · 1.42	0.81 · 1.00 · 1.75	0.53 · 0.77 · 1.13

Toy (home view)	Mid base budget 140,000 splats, 2x phone (September 29, 2026)	High base budget 200,000 splats, 2x phone	3x phone, pixel-ratio cap lifted
Marble	0.81 · 0.69 · 1.13	0.78 · 0.83 · 1.41	0.55 · 0.48 · 0.87
White blood cell	0.77 · 1.28 · 1.64	0.77 · 1.39 · 1.02	0.54 · 0.75 · 1.15
Chladni plate	0.96 · 3.85 · 9.59	0.98 · 5.51 · 10.75	0.75 · 1.48 · 5.39

Each cell is edge · speckle · shimmer, and lower is better for all three. Edge is how many screen pixels an edge takes to go from 10 to 90 percent; speckle is the noise over flat areas; shimmer is how much fine detail flickers frame to frame. Six of the 14 toys measured; the rest are in `docs/lab/SHARPNESS.md`. The splat counts are base budgets, not counts of splats drawn, and the cap on the screen's pixel ratio was 2 on Mid and High that day; it is 3 now.

What it did not test. One pair of budgets (140,000 and 200,000), one phone size, one renderer (software, not a real phone's graphics chip), and the toys as they were on that day, before later changes. The Kit lab's larger sweep is in Chapter 2, under How many splats?.

If it were chosen differently. A bigger budget costs memory and drawing time on a weak device. A smaller one makes shapes coarser. With labs on, a Detail slider now sets the count directly, from 60,000 to 400,000, beside the tiers (`docs/lab/DETAIL-SLIDER.md`); the tiers themselves have not changed.

Where it lives: `PROFILES` in `src/generators.js` ; `detectProfile` in `src/player.js` ; `timeFrame` in `src/stage.js` ; the measurements in `docs/lab/SHARPNESS.md` .

Why this number? One color per splat

OUR CHOICE A captured file can carry more; the kit does not.

A captured file can store how each splat's color changes with the direction you look from (spherical harmonics), because training sees every splat from many angles. A kit splat is placed by a recipe, so there are no photos to learn that change from. The kit keeps one color per splat, and the harmonics of a trained file are dropped when it is used as a kit part.

If it were chosen differently. A recipe can already fake two such looks: `glint` brightens a splat as the camera moves, and `rim` changes glass's opacity with the view. A real shine that moves with the view now exists as a labs experiment, on only

when a link or the address asks for it with labs on (`docs/lab/GLOSS.md`). It uses the kit's own data and shader, with no change to the engine. Real-phone checks are still open.

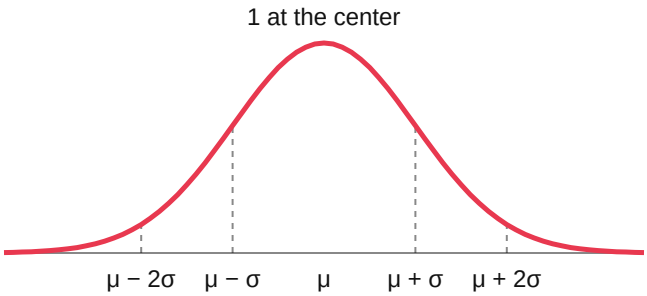
Where it lives: `src/packs/fidelity.js` (its opening comment), `src/kit.js` , and the effect list in `src/effects.js` .

The Gaussian: a bell in three dimensions

The "Gaussian" in Gaussian splat is the bell curve, named after the mathematician Carl Friedrich Gauss. In one data dimension, a bell centered at μ with width σ (sigma, the standard deviation) is

$$G(x) = \exp(-(x - \mu)^2 / 2\sigma^2)$$

The data here are one-dimensional: one number, x, goes in, and one number, G, comes out. The drawing below needs two dimensions to show them, one for x and one for G.

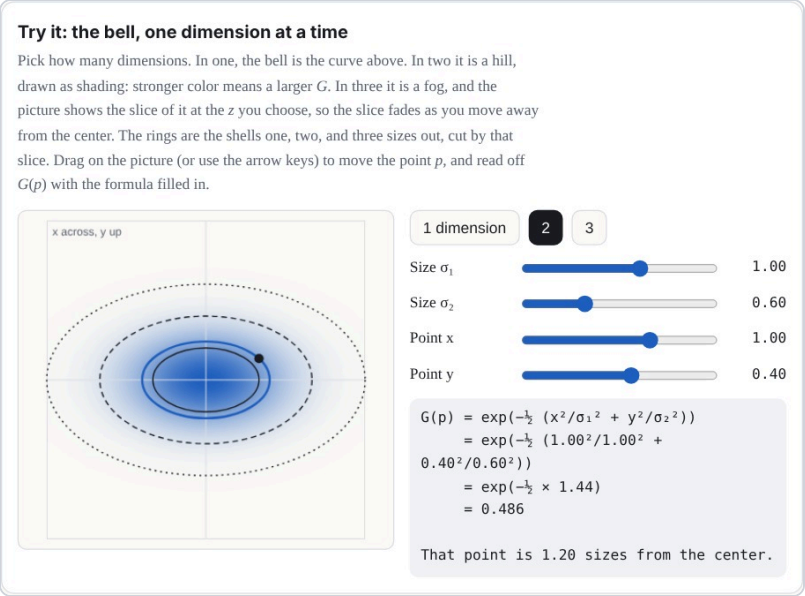


The bell curve. At one σ from the center it has fallen to about 0.61, at two σ to about 0.14, and at three σ to about 0.01, so a splat fades out within about three of its sizes.

In three dimensions, the width becomes a 3-by-3 matrix Σ (capital sigma), called the **covariance** [4], and the squared distance becomes a quadratic form. For a point p in space, the splat's strength there is

$$G(p) = \exp(-\tfrac{1}{2}\,(p - \mu)^{\mathrm{T}}\,\Sigma^{-1}\,(p - \mu))$$

Where G is the same value, you are on the same "shell" around the center, and those shells are ellipsoids: stretched and turned spheres.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/splat.html#demo-bell

Two words in that formula: the inverse and the quadratic form

The inverse Σ^{-1} is the matrix that undoes Σ. And a **quadratic form** is an expression where every term has two factors of the input, like $ad_1^2 + 2bd_1d_2 + cd_2^2$. In two dimensions, with $d = (d_1, d_2) = p - \mu$, and Σ^{-1} with 4 and 1 on its diagonal (and zeros elsewhere), the form $d^{\mathrm{T}}\,\Sigma^{-1}\,d$ is $4d_1^2 + d_2^2$. Setting it equal to 1 draws an ellipse that reaches ½ along the first axis and 1 along the second. A quadratic form is what turns a point's offset from the center into one number for how far out it is, measured in the splat's own widths. In three dimensions, the same idea draws an ellipsoid [5].

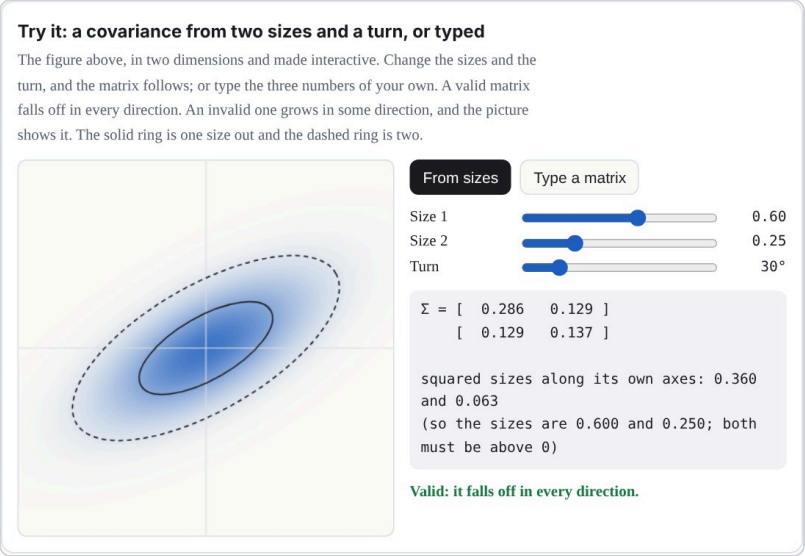
Approximation: the drawn bell is finite. The ideal G never reaches zero. The site's renderer cuts each splat off where G has fallen to e^{-4}, about 0.02 (a little under 3 sizes out), and rescales the falloff so it reaches exactly 0 there. A labs-only setting can swap in a sharper falloff for one toy. So the equations here describe the ideal splat, and the screen shows a close, cut-off copy of it.

The covariance, built from sizes and a rotation

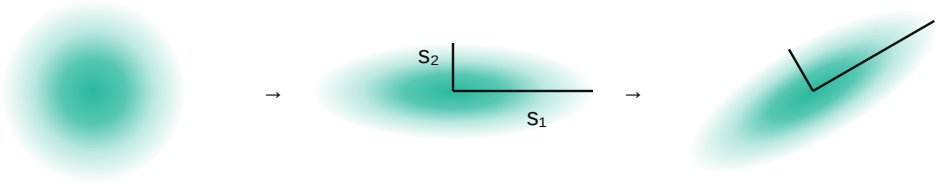
Not every 3-by-3 matrix is a valid covariance. It has to be symmetric, and it has to be positive definite (the bell must fall off in every direction). Instead of picking its nine numbers directly, a splat builds it from its three sizes and its rotation:

$$\Sigma = R\,S\,S^{\mathrm{T}}\,R^{\mathrm{T}}, \quad \text{where} \quad S = \begin{pmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{pmatrix}$$

and R is the rotation matrix of the quaternion q. Read it right to left, as a recipe for a shape: start with a round ball of size 1, stretch it by s_1, s_2, and s_3 along x, y, and z, then turn it by R. Any sizes above zero and any rotation give a valid covariance, which is exactly what you want when a computer is nudging millions of numbers.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/splat.html#demo-covariance



A covariance from sizes and a rotation, drawn in two dimensions: a round splat is stretched along its own axes, then turned.

A worked example: a covariance from sizes 0.3, 0.1, and 0.1

Take sizes 0.3, 0.1 and 0.1, turned 30° about the z axis. Then SS^{T} has 0.09, 0.01, and 0.01 on its diagonal, and multiplying out $R\,SS^{\mathrm{T}}\,R^{\mathrm{T}}$ gives

$$\Sigma \approx \begin{pmatrix} 0.070 & 0.035 & 0 \\ 0.035 & 0.030 & 0 \\ 0 & 0 & 0.010 \end{pmatrix}$$

The top-left number is $0.09\cos^2 30° + 0.01\sin^2 30° = 0.07$, and the ones beside it are $(0.09 - 0.01)\sin 30°\cos 30° \approx 0.035$. Those off-diagonal numbers are the **tilt**: they are zero when the splat's axes line up with x, y, and z, and usually not zero when they do not. (Usually, because a splat that is round in a plane has no preferred axes there, so turning it within that plane changes nothing.) (In this manual, **rotation** means a splat's own turn, and "Tilt lock" is the name of the camera control in Chapter 5.)

From 3D to the screen

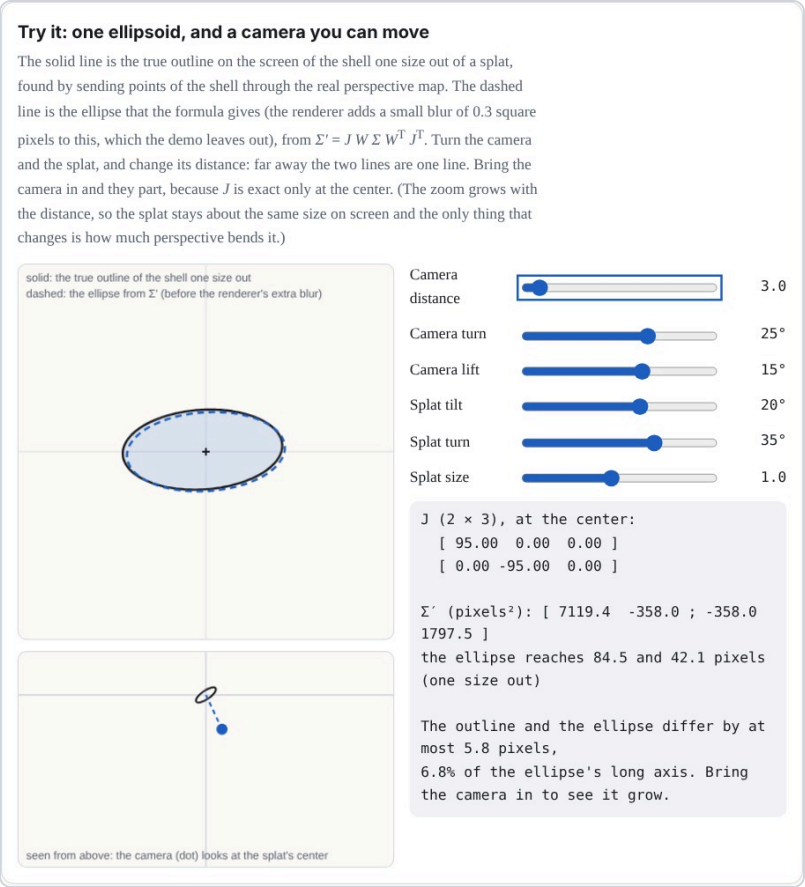
To draw a splat, the renderer works out what its bell looks like from the camera. It first turns the covariance into the camera's view, with the camera's rotation W. The perspective of a camera is not a straight-line map, but close to one point it nearly is, and its best straight-line stand-in there is a matrix J (the Jacobian of the projection). Such a stand-in is called **affine**: a straight-line map plus a shift [6]. Then the splat's covariance on the screen is

$$\Sigma' = J\,W\,\Sigma\,W^{\mathrm{T}}\,J^{\mathrm{T}}$$

keeping only its top-left 2-by-2 block. That 2-by-2 matrix is a flat bell on the screen: an ellipse. Its eigenvectors are the directions of the ellipse's axes, and the square roots of its eigenvalues are how far it spreads along them. This trick comes from EWA splatting (Zwicker, Pfister, van Baar, and Gross, 2001) [2].

That is the first-order footprint, and a real renderer draws a little more than it. To keep a tiny splat from shrinking to less than a pixel (and flickering), splat renderers add a small blur on the screen: the original 3D Gaussian splatting code adds 0.3 square pixels to each diagonal entry of the 2-by-2 matrix, and so does Splashery's renderer, which is PlayCanvas's. In its vendored shader that blur is always on: it adds 0.3 to both diagonal entries, and it keeps the smaller eigenvalue at least 0.1. It also skips splats that come out under about 2 pixels on the screen (the cull settings in `src/sharpness.js` explain how). A separate, optional setting, the engine's anti-aliasing (off unless a labs switch turns it on), does something else: it lowers each

splat's opacity to make up for the energy that blur spread out, using the ratio of the determinants before and after. So the demo below shows Σ′ itself, the unfiltered geometry; the pixels on your screen have that extra blur on top, whichever way the option is set.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/splat.html#demo-projection

> **Approximation: affine around the center.** An affine map is a straight-line map plus a shift. *J* is exact only at the splat's center, where the straight-line stand-in touches the true perspective map. Across the rest of the splat the real map bends away from it, so the ellipse is the first-order answer, not an exact image of the 3D bell.

Sorting and blending, back to front

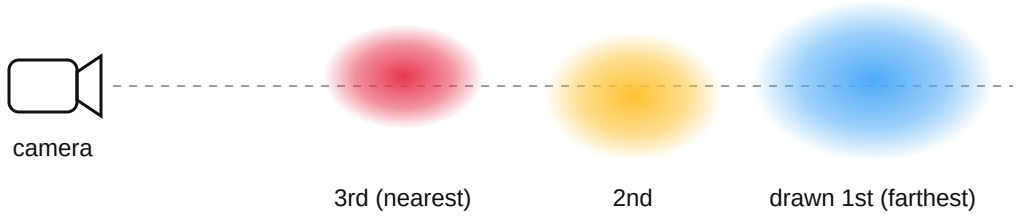
Splats are see-through at their edges, so the order they are drawn in matters. The splats are sorted by their centers' depth along the camera's view, and painted from the farthest to the nearest. At each pixel, a splat's strength there is its opacity times its bell, $a = \alpha\, G'$, and it is laid over what is already there:

$$C \leftarrow a\,c + (1 - a)\,C$$

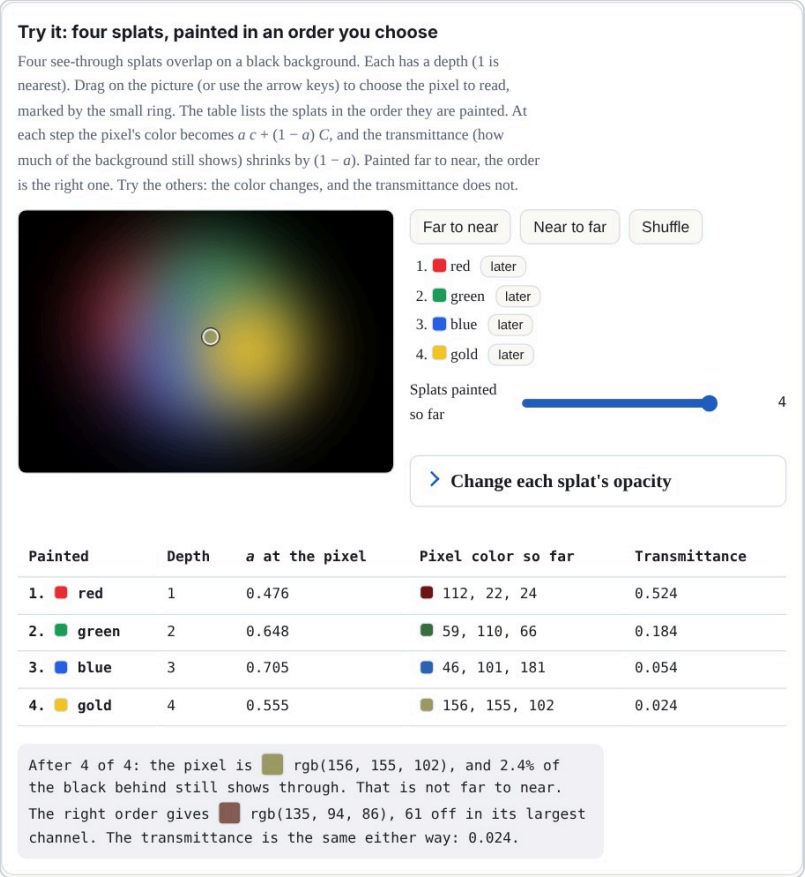
where c is the splat's color and C the pixel's color so far. Written out for all the splats at once, counting from the nearest ($i = 1$) to the farthest, the pixel's color over a black background is

$$C = \Sigma_i\, c_i\, a_i \prod_{j<i} (1 - a_j)$$

Each splat shows as much as the splats in front of it let through. This "over" operation is from computer graphics' classic paper on compositing [3]. It is how the 2023 paper that started the craze, "3D Gaussian Splatting for Real-Time Radiance Field Rendering" (Kerbl, Kopanas, Leimkühler, and Drettakis) [1], draws its scenes, and it is how Splashery blends its splats (the cover lists the exceptions).



Splats are painted from the back to the front, each one laid over the ones behind it.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/splat.html#demo-blend

Why the order matters in the toys. By default, Splashery sorts the splats in the pose a toy was built in, so a piece that moves keeps the order it had at rest. A recipe can request `out.resortPose` to sort moving parts and pages in their current pose. Turn a solid thing more than a quarter turn and its far side can be drawn over its near side. The toys work around it: the Splat equation toy, for one, builds its shape at twelve moments and morphs from the copy at the start of the current interval to the next (Chapter 3).

A worked example of blending, with a background

Take one color channel. The background is 0.2. The far splat (drawn first) has color 0.9 and strength $a = 0.5$ at this pixel. The near splat (drawn second) has color 0.1 and $a = 0.4$. After the far splat: $C = 0.5 \times 0.9 + 0.5 \times 0.2 = 0.55$. After the near one: $C = 0.4 \times 0.1 + 0.6 \times 0.55 = 0.37$. The same number comes from the sum above with the background added: $0.1 \times 0.4 + 0.9 \times 0.5 \times 0.6 + 0.2 \times 0.6 \times 0.5 = 0.37$. The share of the background that still shows, the **transmittance**, is $0.6 \times 0.5 = 0.3$.

One caution about the numbers. 0.37 is the arithmetic of one channel in one consistent blending space: the formula does not care what the 0 to 1 numbers mean,

as long as every number in the sum uses the same meaning. A screen's red, green, and blue numbers are usually *encoded* (squeezed so that dark shades get more steps), and blending encoded numbers is not the same as blending the light itself. So read 0.37 as the result of the sum, not as a measured amount of light and not as the final pixel on any particular screen.

Approximation: sorting uses centers. Each splat gets one depth, the depth of its center, so the order is per splat, not per pixel. Two big splats that overlap in depth can be drawn in the wrong order where they cross. This is why recipe authors keep layers apart (the recipe guide says splats "sort by their centers, so closer layers mix in stripes").

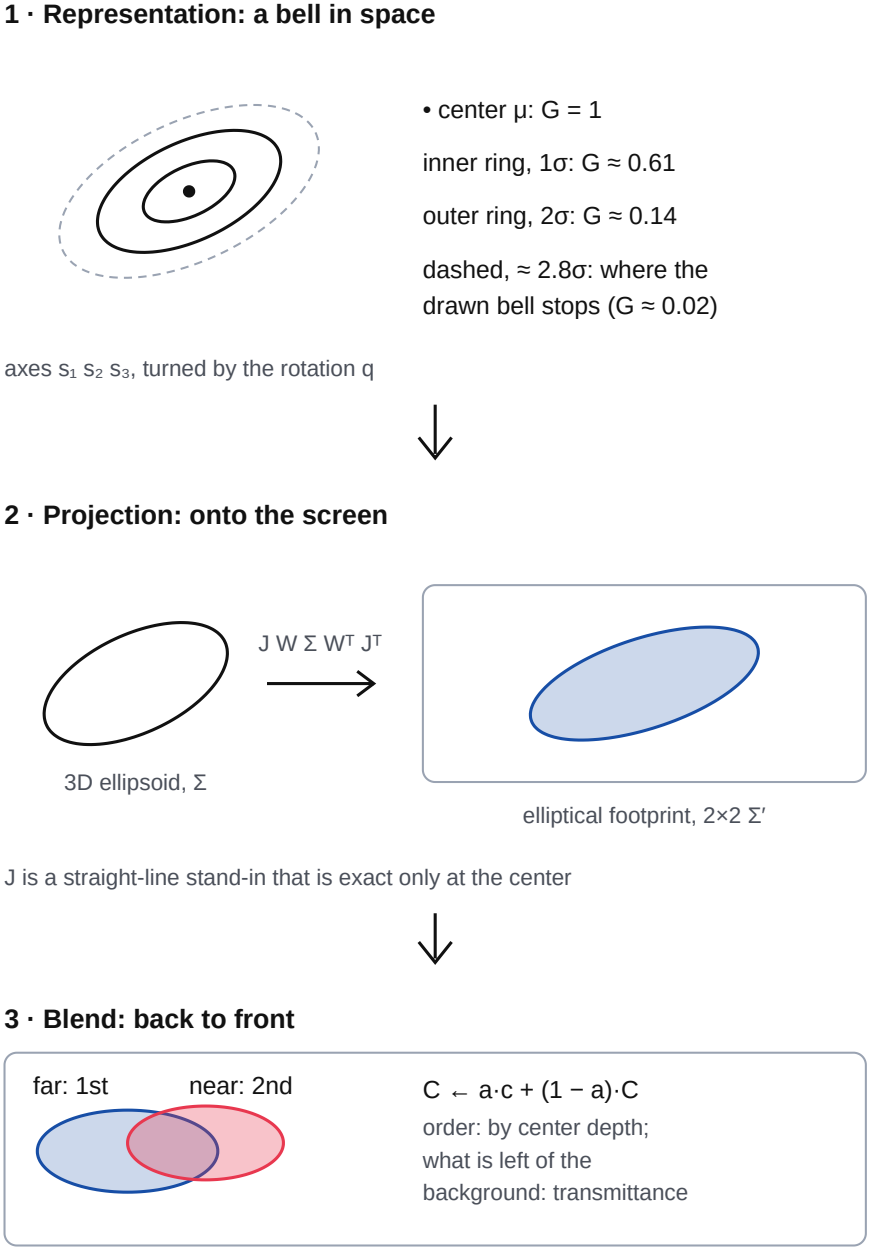


Figure 2. From a splat to a pixel. **1** The ideal bell has equal-weight shells; the drawn one is cut off. **2** The camera flattens the ellipsoid into an ellipse, using a straight-line stand-in for the perspective. **3** The ellipses are blended back to front, each over what is already there.

CHAPTER 2

How a toy is built

Nobody places Splashery's splats by hand. Most toys are **recipes**: short programs that describe a toy with a kit of shapes, and the kit fills them with splats when you open the toy. A recipe is a few kilobytes of code instead of megabytes of splats, and because every random choice comes from a seed, it builds the same splats with the same seed, options, and budget.

The **kit** is Splashery's toy-building library. A recipe names shapes, colors, and moving parts; the kit fills them with splats in the browser, spreading the budget evenly by area. A **kit toy** is built this way. A scan or capture is loaded from a file instead.

Recipes are written in JavaScript. This chapter is a tour, so you can read one; the complete guide is docs/PACKS.md in the source code [10].

Two places a toy lives

A toy's name and shelf go in a **catalog** entry (`src/toys.js`), and the toy itself in a **recipe** in a pack file (`src/packs/…js`):

```
{
  id: "little-windmill",       // lower-case words joined by hyphens
  label: "Little windmill",    // its name on the shelf
  category: "objects",         // which shelf
  kind: "kit",
  pack: "my-pack",             // loads src/packs/my-pack.js
}
```

A recipe has six basic parts, all optional except `build` (Chapter 5 adds `pictures` , `input` , and `tiltLock` , and PACKS.md has more, such as `grab` and `drag` for touch play):

Show the table: the six parts of a recipe

`build(k, o)`	Describes the toy with the kit `k` . `o` holds the chosen options.
`options`	Choices that rebuild the toy: a color, a style, a number of petals. They are saved in links.
`controls`	Live values from 0 to 1: sliders, toggles that ease on and off, and pulses that jump to 1 and fall back.

`action`	Which control a tap on the toy fires.
`drive(t, c, out)`	Runs every frame: from the time `t` and the controls `c`, it sets how the parts move.
`alive`	`true` when the toy moves by itself, so frames keep coming.

Shapes from the kit

`k.sphere(r)`, `k.ellipsoid(a, b, c)`, `k.box(w, h, d)`, `k.roundedBox(w, h, d)`, `k.cylinder(r, h)`, `k.cone(r0, r1, h)`, `k.torus(R, r)`, `k.disc(r)`, `k.lathe(profile)` (a vase or a bottle, turned around an axis), `k.tube(curve, radius)` (a stem, an arm, a knot), `k.param((u, v) => [x, y, z])`, any surface you can write in two variables (the same idea as Chapter 3; here *u* and *v* each run from 0 to 1), and `k.radial((dir) => r)`, a blobby surface given by its distance from the middle in every direction. Nothing is drawn until you add a shape:

```
k.add(k.sphere(0.2), { pos: [0, 1, 0], color: "#ffcc33", weight: 2 });
```

The kit shares the splats out between the shapes by their surface area, so the toy is evenly dense; `weight` gives a small shape more (or `share` a fixed fraction of all the splats). For free-form splats, like stars, sparks, and crumbs, `k.cloud` places them one by one. After `build`, the kit centers the toy and scales it to fit the view, so you can model in any units.

How many splats? What a bigger budget does

The kit spreads a budget of splats over a toy's surface (see Chapter 1 for the four numbers). It is natural to think more splats always look better. The Kit lab measured it (`docs/lab/BUDGETS.md`, October 10, 2026). It drew the same toy from the same view on the same simulated 390 by 844 phone at five budgets, changing only the count. The table compares 140,000 (the Mid tier) with 200,000 (the High tier) for 24 views of 22 toys.

These counts are **base budgets**: the number a tier asks for before the toy's own density. A toy with density 2 asks for twice as many, up to the tier's most (240,000 at Mid and 300,000 at High, from `PROFILES` in `src/generators.js` and `splatBudget` in `src/player.js`). The American football, the hockey puck, and the marble have density 2, so at the Mid and High base budgets they were asked

for 240,000 and 300,000 splats, not 140,000 and 200,000. The small line under each toy in the table gives the splats it actually got at the lowest and highest base budgets (from the document), so the table is about what a bigger base budget does, not about a fixed count per toy.

What the columns mean

- **Edge** is how many screen pixels the strongest edges take to go from 10 to 90 percent of the way between the two colors. A sharp edge is narrow, so lower is sharper.
- **Speckle** is how far pixels on the flatter half of the toy differ from the middle of their neighbors: grit on a smooth surface. Lower is cleaner. Painted grain, like the oak's bark, also counts, because it looks the same at phone size.
- **Shimmer** is how much the fine detail changes from one frame to the next while the toy turns a little. Lower is steadier.
- **Area** is how much of the screen the toy covers, as a percent of the same view at 400,000 splats. Over 100 percent means thin parts drawn fatter than they are. The table shows base budgets of 60,000 and then 140,000, because that is where the change is.

Same toy, same view, same phone, at a base budget of 140,000 splats → 200,000. A base budget is the number the tier asks for before the toy's own density; the small line under each toy gives the splats it actually got at the 60,000 and 400,000 base budgets. Lower is better in every column. Worse Better The same On a screen, the table scrolls sideways inside its box.

Toy (view)	Edge (screen pixels)	Speckle	Shimmer	Area, base 60k → 140k
Lamp 60,000 → 399,998 splats drawn at the 60k → 400k base budgets	0.52 → 0.51	**0.37 → 0.39**	**1.57 → 1.65**	101% → 100%
American football 120,000 → 600,000 splats drawn at the 60k → 400k base budgets	0.50 → 0.48	**0.61 → 0.70**	**1.49 → 1.60**	100% → 100%
Hockey puck 120,000 → 600,000 splats drawn at the 60k → 400k base budgets	**0.27 → 0.29**	0.24 → 0.21	1.01 → 0.99	102% → 101%
Klein bottle 48,000 → 320,000 splats drawn at the 60k → 400k base budgets	0.56 → 0.53	**0.52 → 0.90**	**1.12 → 1.33**	100% → 99%

Toy (view)	Edge (screen pixels)	Speckle	Shimmer	Area, base 60k → 140k
Clock 60,000 → 400,000 splats drawn at the 60k → 400k base budgets	0.51 → 0.49	**0.49 → 0.54**	**1.64 → 1.90**	101% → 100%
Penguin 60,001 → 400,002 splats drawn at the 60k → 400k base budgets	0.51 → 0.49	**0.42 → 0.46**	**2.92 → 3.13**	–
Snow globe 57,539 → 383,576 splats drawn at the 60k → 400k base budgets	0.44 → 0.44	**0.20 → 0.23**	**0.71 → 0.84**	102% → 101%
White blood cell 60,000 → 400,001 splats drawn at the 60k → 400k base budgets	0.54 → 0.53	**0.72 → 0.82**	**0.84 → 1.02**	101% → 100%
Ocean liner 53,356 → 355,780 splats drawn at the 60k → 400k base budgets	0.46 → 0.44	**0.31 → 0.35**	**1.69 → 2.09**	106% → 104%
Marble 120,001 → 600,001 splats drawn at the 60k → 400k base budgets	0.46 → 0.45	**1.43 → 1.65**	**2.63 → 2.96**	100% → 100%
Sailboat 60,000 → 400,002 splats drawn at the 60k → 400k base budgets	0.50 → 0.46	**0.51 → 0.59**	**2.26 → 2.93**	103% → 103%
Toy piano 60,189 → 400,004 splats drawn at the 60k → 400k base budgets	0.60 → 0.56	**0.22 → 0.25**	**1.93 → 2.32**	111% → 105%
Your book 113,837 → 446,581 splats drawn at the 60k → 400k base budgets	**0.52 → 0.55**	0.00 → 0.00	1.17 → 0.46	–
Chladni plate 51,398 → 149,874 splats drawn at the 60k → 400k base budgets	0.66 → 0.66	0.26 → 0.26	6.02 → 6.01	100% → 100%
Bicycle 59,097 → 393,919 splats drawn at the 60k → 400k base budgets	0.50 → 0.49	**0.89 → 0.99**	**6.00 → 6.52**	118% → 107%
Eiffel tower 60,002 → 400,002 splats drawn at the 60k → 400k base budgets	0.58 → 0.55	**0.20 → 0.24**	**0.86 → 1.19**	109% → 104%

Toy (view)	Edge (screen pixels)	Speckle	Shimmer	Area, base 60k → 140k
Crystal lattice 59,872 → 399,968 splats drawn at the 60k → 400k base budgets	0.57 → 0.54	**0.27 → 0.31**	**1.49 → 1.83**	111% → 105%
Knot 60,000 → 400,000 splats drawn at the 60k → 400k base budgets	0.53 → 0.52	**0.48 → 0.58**	**1.54 → 1.87**	103% → 101%
Torus knot 60,000 → 400,000 splats drawn at the 60k → 400k base budgets	0.53 → 0.52	**0.29 → 0.34**	**1.10 → 1.25**	98% → 98%
Oak 59,792 → 398,257 splats drawn at the 60k → 400k base budgets	0.58 → 0.57	**0.40 → 0.51**	**0.82 → 1.03**	105% → 102%
Chess set 58,249 → 387,829 splats drawn at the 60k → 400k base budgets	0.58 → 0.52	**0.14 → 0.15**	**2.44 → 3.09**	106% → 103%
Dna 59,986 → 400,020 splats drawn at the 60k → 400k base budgets	0.51 → 0.50	**0.50 → 0.57**	**4.06 → 4.53**	–
Bicycle (close up, 0.5) 59,097 → 393,919 splats drawn at the 60k → 400k base budgets	0.53 → 0.52	**0.36 → 0.42**	**2.78 → 3.25**	118% → 107%
Chess set (close up, 0.6) 58,249 → 387,829 splats drawn at the 60k → 400k base budgets	0.60 → 0.54	**0.08 → 0.09**	**1.50 → 2.01**	103% → 101%

All five budgets across all the views: the middle value of each toy's change against its own result at a base budget of 140,000 splats.

Base budget	Edge	Speckle	Shimmer	Thin parts' area
60k	+11%	−23%	−36%	+1.2%
140k	–	–	–	–
200k	−2%	+14%	+18%	−0.8%
280k	−6%	+31%	+39%	−0.8%
400k	−9%	+52%	+61%	−1.2%

What it shows

- **Going from 140,000 to 200,000, speckle rose in 21 of the 24 views and shimmer rose in 21 of them** (the counts are from the table). Across all the views the middle change was 14 percent more speckle and 18 percent more shimmer. Each splat gets smaller as the count goes up, so the fine detail in a pixel comes from more, smaller splats that move in and out of it as the view turns.
- **Edges barely changed.** The most any edge gained between the two budgets was 0.06 of a screen pixel, the middle change was 2 percent, and in two views the edge got a little softer.
- **The one place a bigger budget clearly helps is thin parts.** At 60,000 splats the bicycle's frame and spokes cover 18 percent more of the screen than at 400,000; at 140,000 they are within 7 percent. Solid toys, like the clock or the marble, stay within 1.5 percent of the 400,000 view even at 60,000 splats.
- **No holes open at the lower budgets.** A low budget makes thin parts fatter, not broken.

What it does not show: one renderer (a software one, not a real phone's graphics chip), one phone size, still views with a slow turn, and no frame rate you could trust. Scans (captured toys) are not affected, because they load their own files. The kit lab's recommendation was to keep the four tiers as they are. The full tables, the method, and the limits are in docs/lab/BUDGETS.md and the first round of this question is in `docs/lab/SHARPNESS.md`.

Colors from functions

A color can be a fixed quantity or vector (`"#9c3b2c"` , or `[r, g, b]` with each from 0 to 1; there are no color names), or it can be a function that the kit calls for every splat. The function gets the splat's place `c.p` , its surface direction `c.n` , its surface coordinates `c.u` and `c.v` , and a random number and noise that come from the seed. Return `null` to leave a hole:

```
color: (c) => (c.p[1] > 0.5 ? "#ffffff" : "#3a7d44"), // snow above, grass below
```

Parts that hinge, spin or slide

A **part** is a group of shapes that moves as one solid piece: `k.part("lid", { pivot, axis })` . Every frame, `drive` tells each part where to be: an `angle` about its axis (a lid on a hinge, a wheel that spins), an `offset` (a drawer that slides), a `scale` , or `visible` . A toy may have up to 15 parts. The treasure chest's lid is a part on a hinge.

Why this number? Fifteen parts

OUR CHOICE, NEAR A LIMIT The format of the part number sets the count; the shader's space makes it hard to raise.

Every splat carries the number of the part it belongs to, and that number gets only 4 bits of a packed value, because the same value also holds two flags for the shader. Four bits count 16 things. Number 0 is the body, which every toy has, so 15 parts are left to name. Asking for a sixteenth stops the build with a message. The shader keeps three vector values for each of the 16 parts (a rotation, a pivot, and an offset with visibility), 48 in all.

If it were chosen differently. More parts means repacking the part number so the two flags move, moving the part table out of the shader's small set of uniform values and into a texture, and checking real phones. That is a change to the engine, not to a recipe. The kit has other ways to get many moving pieces: up to 48 tokens (game pieces moved and turned by their own offsets) and up to 96 levers in 6 groups (the piano keys).

Where it lives: `part` in `src/kit.js`; the part number in the shader code of `src/effects.js`; `MAX_TOKENS` in `src/motion.js`.

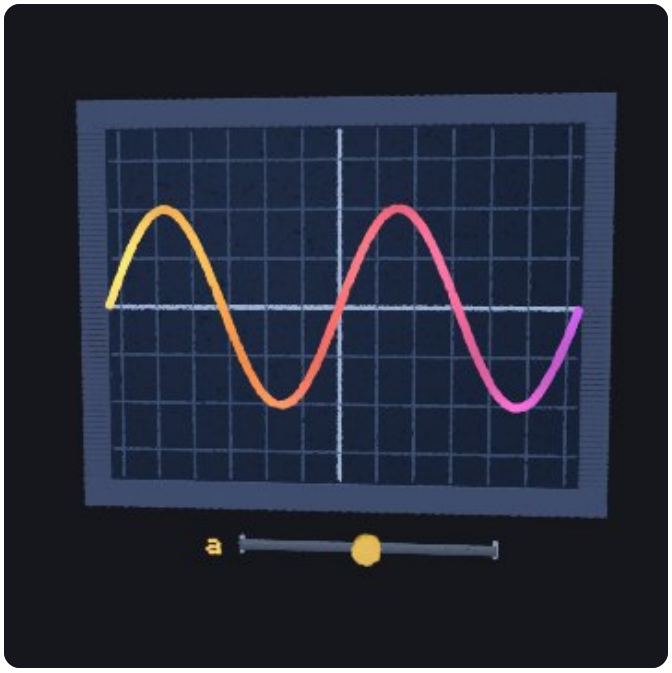
Could a toy use different parts?

EACH RECIPE'S CHOICE

The 15 is a limit on how many parts a toy can have. It comes from the 4-bit part number. Which pieces become parts is up to each recipe. A recipe makes a part with `k.part(name, { pivot, axis })` and puts shapes in it with `part:`. The windmill makes one part for all four sails. A recipe could instead make one part per sail, so each could turn on its own; it would then have used four of its 15.



The treasure chest: its lid is a part on a hinge.



The graph plotter: the a slider is a control.

Behaviors

Some motion is too fine for parts: a flame that flickers, snow that falls, grass that sways. A **behavior** moves each splat on its own on the graphics card: `kind: "flame"`, `"twinkle"`, `"sway"`, `"orbit"`, `"fall"`, `"wave"`, and more. **Morphs** give each splat a second place, `to: (c) => [x, y, z]`, and slide it there as one of four channels goes from 0 to 1. The Splat equation toy's motion is made of morphs.

A small, complete recipe

Here is a whole toy: a little windmill whose sails turn with a Wind slider, and a tap that blows a gust, spinning them once more around.

```
import { mix } from "../src/kit.js";

// A little windmill: a stone tower, a red cap, and four sails that turn in
// the wind. A tap blows a gust that spins them once more around.
export const RECIPES = {
  "little-windmill": {
    alive: true,
    options: [{ key: "sails", label: "Sail color", type: "color", default: "#f2eee4" }],
    controls: [
      { key: "wind", label: "Wind", type: "slider", default: 0.4 },
      { key: "gust", label: "Gust", type: "pulse", ease: 3 },
    ],
    action: { key: "gust", label: "Blow a gust" },
```

```
    drive(t, c, out) {
      // The wind turns the sails; a gust adds one extra turn, eased in and
      // out, so they end where they would have been anyway.
      const g = c.gust > 0 ? 1 - c.gust : 0;
      const extra = 2 * Math.PI * g * g * (3 - 2 * g);
      out.parts.sails = { angle: t * (0.3 + 2 * c.wind) + extra };
    },
    build(k, o) {
      // The tower: whitewashed stone, darker toward the ground.
      k.add(k.cone(0.5, 0.3, 1.6), {
        color: (c) => mix("#8b8f97", "#ece6d8", (c.p[1] + 0.8) / 1.6),
        interior: 0.1,
        core: "#6b5b4b",
      });
      // The cap.
      k.add(k.cone(0.36, 0.02, 0.4), { pos: [0, 1.0, 0], color: "#9c3b2c" });
      // The sails turn together, about an axis that points at you.
      const sails = k.part("sails", { pivot: [0, 0.6, 0.42], axis: [0, 0, 1] });
      k.add(k.sphere(0.07), { pos: [0, 0.6, 0.42], part: sails, color: "#4b3a2a" });
      for (let i = 0; i < 4; i++) {
        const a = (i * Math.PI) / 2;
        k.add(k.box(0.16, 0.9, 0.02), {
          pos: [-0.5 * Math.sin(a), 0.6 + 0.5 * Math.cos(a), 0.44],
          rot: [0, 0, (a * 180) / Math.PI],
          part: sails,
          color: o.sails,
        });
      }
    },
  },
};
```

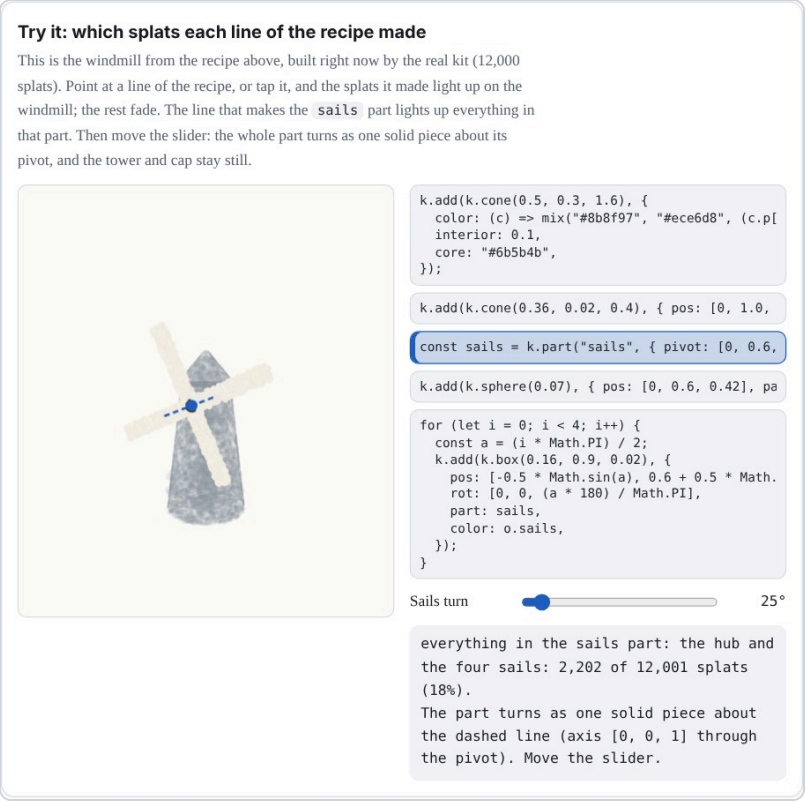


This recipe, built by the kit.

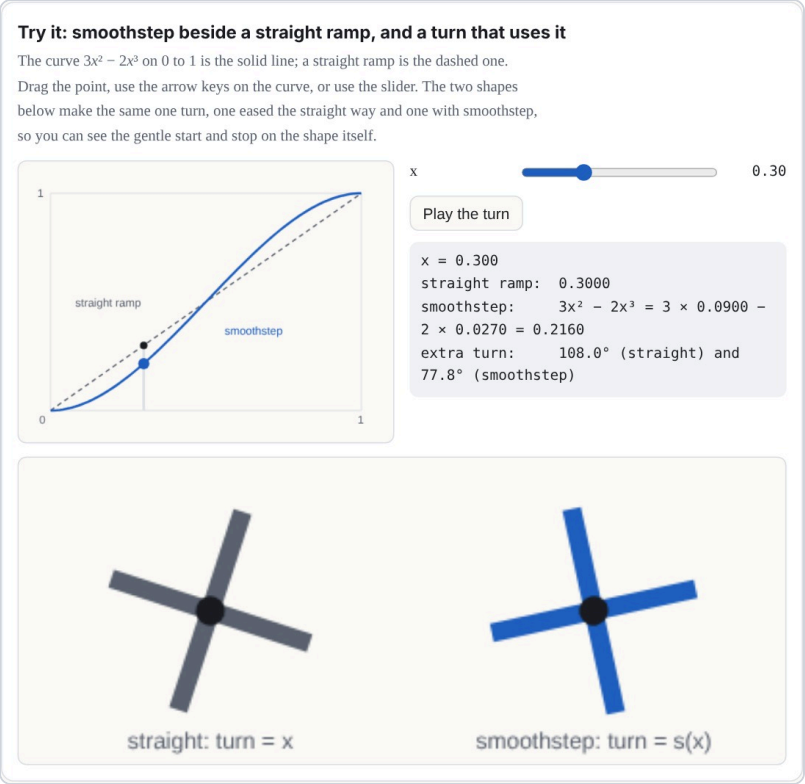


The Windmill on the shelf, a bigger toy built the same way.

Read it from the bottom up. `build` adds a cone for the tower (its color a function of height), a smaller cone for the cap, and a part called "sails" with a hub and four boxes, each turned a quarter turn more than the last. `drive` turns the sails part: steadily with the wind, plus one extra turn while the gust pulse runs, eased with the **smoothstep** curve $3g^2 - 2g^3$ (an S-shaped ramp that rises from 0 to 1 and is flat at both ends [7]) so it starts and ends gently. A full extra turn lands the sails where they would have been anyway, so nothing jumps when the gust ends.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/toy.html#demo-windmill



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/toy.html#demo-smoothstep

Run it yourself

You can run the windmill on your own computer in five steps. You need a copy of the Splashery source (download it from GitHub) and Python 3.

1. **Save the recipe as a pack.** Copy the module above into `src/packs/little-windmill.js`, or copy manual/example-recipe.js. Change its first line to `import { mix } from "../kit.js";`. A pack lives in `src/packs/`, so the kit is one folder up. (The copy in `manual/` says `../src/kit.js` because it sits beside `src/`, not in it.)
2. **Add its catalog entry.** In `src/toys.js`, find the line `export const TOYS = [` and paste this right after it:

   ```
   {
     id: "little-windmill",
     label: "Little windmill",
     category: "objects",
     kind: "kit",
     pack: "little-windmill",   // loads src/packs/little-windmill.js
   },
   ```

   The `id` must match the key in the pack's `RECIPES`.

3. **Serve the folder.** In the source folder, run `python3 -m http.server 8000 --bind 127.0.0.1`. The page needs a server because browsers will not load ES modules from a file address.
4. **Open the toy.** Go to `http://127.0.0.1:8000/embed/index.html?toy=little-windmill`. The embed page opens a catalog toy by its id. The windmill turns by itself.
5. **Tap the windmill.** The tap runs the recipe's action, the **Blow a gust** pulse: the sails spin one extra turn, easing in and out, and end where they would have been.

The same steps, as a prompt. If you work with a coding assistant, paste this into it, from inside your copy of the source folder:

```
In this copy of the Splashery source, please do exactly these steps and nothing else:
1. Create src/packs/little-windmill.js as a copy of manual/example-recipe.js, but change its
   first line to: import { mix } from "../kit.js";
2. In src/toys.js, right after the line "export const TOYS = [", add this entry:
   { id: "little-windmill", label: "Little windmill", category: "objects", kind: "kit",
     pack: "little-windmill" },
3. Start a local server in the source folder: python3 -m http.server 8000 --bind 127.0.0.1
4. Tell me to open http://127.0.0.1:8000/embed/index.html?toy=little-windmill and what to
   expect: the sails turn by themselves, and a tap spins them one extra turn.
```

> The Splat equation toy's text box takes equations (Chapter 3), not JavaScript recipes.
> A recipe has to be a module in `src/packs/`, as above.

CHAPTER 3

The splat equation language

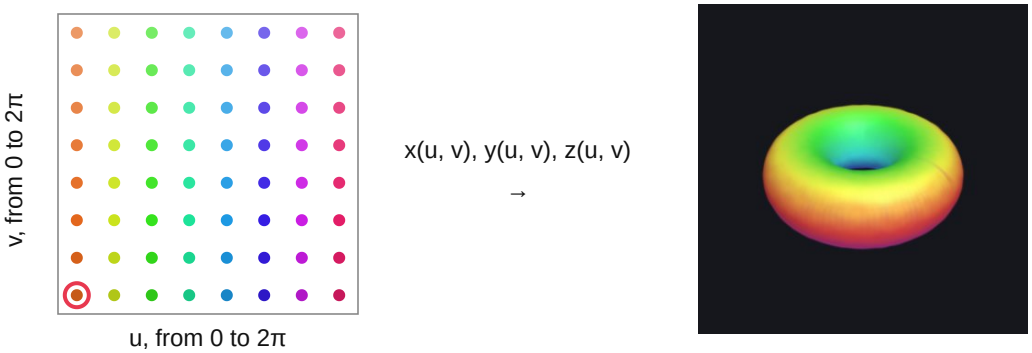
Recipes are powerful, but they are code, and code from a stranger can do things you did not ask for. The **Splat equation** toy on the math shelf gives you a smaller language that is only math. You write where every splat goes and what color it is, as equations, and nothing else can happen. That is why a link with your equations in it is safe to share.

Open it with any link in the gallery (Chapter 4). Type in the box under **Your own splat program** in the Toy tab and press **Splat it**. The Program list has eight famous shapes to start from.

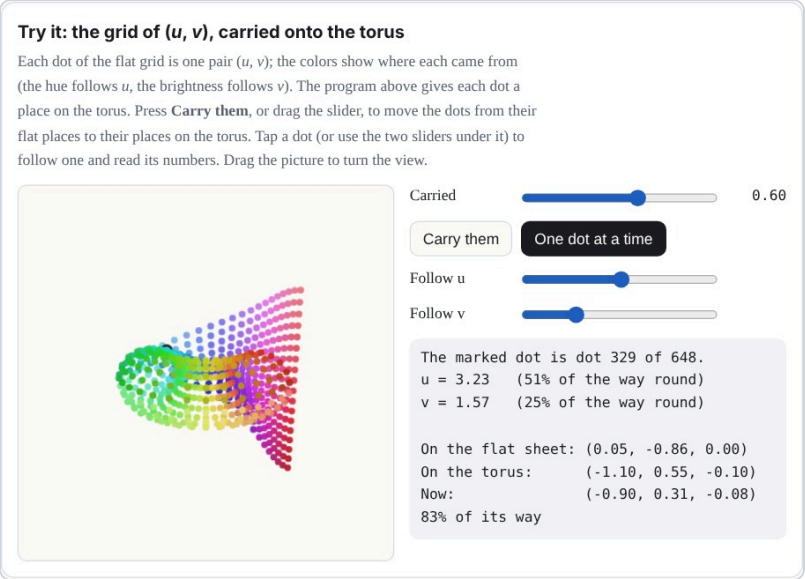
The idea: every splat gets its own u and v

The toy lays out its splats on a grid of two numbers, *u* and *v*, each running over a range you choose. Think of them as the coordinates of a flat sheet of points, laid out before the equations bend it into shape. They are the parameters of a parametric surface [9], not a substitution: nothing is replaced by *u* or *v*; each splat simply has its own pair. Your equations turn each splat's (*u*, *v*) into a place in space, (*x*, *y*, *z*). If *u* goes around a circle and *v* around a smaller circle, the grid wraps into a torus:

```
x = (2 + cos v)·cos u
y = sin v
z = (2 + cos v)·sin u
u = 0 .. 2pi
v = 0 .. 2pi
```



Each dot on the grid is one splat with its own (*u*, *v*). The equations carry every dot to its place on the torus.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/language.html#demo-uvtorus

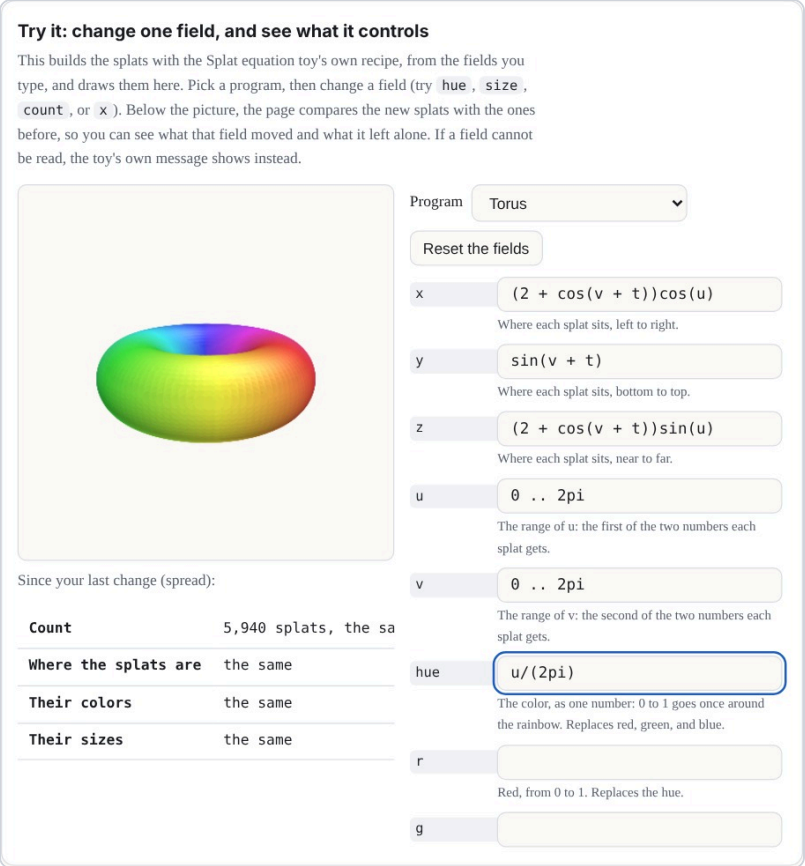
The fields

A program is a list of **fields**, each written `name = value`. You can type one field or several, split by `;` (a text file can put one on each line). You do not have to retype everything. A field you leave out keeps what is already showing: if you type only a new `hue`, the shape stays and only the color changes.

Show the table: the fields of a program

Field	What it is	If left out
`x`, `y`, `z`	Where each splat goes, as equations in *u*, *v*, and *t*. *y* points up.	Needed
`u`	The range of *u*, like `0 .. 2pi`.	Needed
`v`	The range of *v*.	`0 .. 1`
`hue`	Color as a place on the color wheel: 0 is red, 1/6 yellow, 1/3 green, 1/2 cyan, 2/3 blue, 5/6 magenta, and 1 is red again. Any number works; it wraps around every whole turn, so −0.1 gives the same hue as 0.9.	Use *r*, *g*, *b*
`r`, `g`, `b`	Color as red, green, and blue, each from 0 to 1. Setting any of them turns *hue* off; setting *hue* turns them off.	0.5 each (gray)

Field	What it is	If left out
`size`	How big each splat is, in the same units as *x*, *y*, and *z*: about the σ of its bell. It may use *u*, *v*, and *t*.	`0.05`
`count`	How many splats: 100 to 10,000.	`4000`
`spread`	`grid` spaces the splats evenly; `random` scatters them at random over the ranges (a galaxy, a firework).	`grid`



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/language.html#demo-fields

Why this number? Twelve fields

OUR CHOICE The toy chose which fields; saved links limit how many a toy can keep.

The toy's program is saved in its link as options, and a link keeps up to 64 option keys per toy (16 until October 17, 2026). The twelve fields, plus the program picker, the splat style (Solid, Fine, or Dots), the Light and shade switch, and the Moments of t choice, come to 16, so a fully filled toy leaves 48 keys to spare. (The labs switch

only hides the toy from the shelf; it does not take the Moments of t definition away, so a link that opens the toy directly still counts all 16.)

If it were chosen differently. More fields means more option keys. Up to 64 fit a link; past that, either the limit rises again (a change to how every saved link is read, which must keep old links loading) or fields share a key.

Where it lives: `normalizeOptions` in `src/state.js`; `FIELDS` in `src/packs/splat-equation.js`.

Could the equation toy have different fields?

NOT TESTED YET Nothing below is built.

Yes. Each field you add costs one of the 64 option keys a saved link keeps, and the toy uses 16, so 48 new fields would fit before the limit had to rise or fields had to share a key. Candidates:

- **Opacity**, so a splat could be see-through.
- **A stretch and a turn for each splat**, which would make an oriented, flattened splat instead of a round one.
- **Glow**, a brightness that does not depend on the lighting.
- **A part for each point**, so each point could move on its own. This is the costly one: toys are limited to 15 parts.

Flexibility

The reader is forgiving about how you write a program:

- **Alternative names.** `color` for `hue`; `red`, `green`, and `blue` for `r`, `g`, and `b`; `splats` for `count`; `x(u,v,t)` for `x` (and likewise `y(u,v,t)` and `z(u,v,t)`); also `colour` and `n`, which the code accepts.
- **Three ways to write a range.** `0 .. 2pi`, `0 to 2pi`, or `0, 2pi`. (Three dots, `0 ... 2pi`, work like two.)
- **Capitals do not matter.** Names are not case-sensitive: `Hue` is `hue`.
- **Notes.** Anything after a `#` on a line is a note, and the reader skips it.

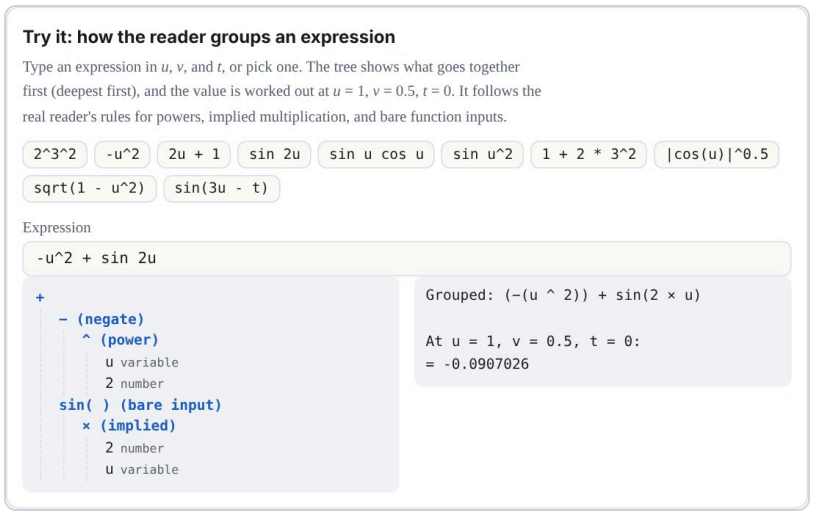
Expressions

The equations are read by the same small, safe reader as the Graph and Surface plotters. It understands:

- **Numbers**: `2`, `0.5`, `.25`.
- **Variables**: *u*, *v*, and *t*. Range ends and the count are numbers only.

- **Constants**: `pi` (or π) and `e`.
- **Operators**: `+ − × ÷ ^`, written `+ - * / ^` (and `**`, `·`, `²`, `³`), with brackets `( )`, `[ ]`, and `{ }` (the reader treats all three kinds alike), and `|u|` for absolute value.
- **Functions**: `sin cos tan asin acos atan sinh cosh tanh exp log` (natural; `ln` too), `log10 sqrt` (or √), `abs floor`, and `min` and `max` of two or more values. The inverse trig functions also go by `arcsin`, `arccos`, and `arctan`.
- **Shortcuts**: `2u` means 2 × *u*, and `3sin(u)` and `(u + 1)(v − 1)` multiply. A function can also skip its brackets when its input is a short product: a run of numbers and names, each with an optional power. The input stops at the next function, at a bracket, or at a plus or minus. So `sin 2u` is sin(2*u*), `sin u cos u` is sin(*u*) × cos(*u*), and `sin u^2` is sin(*u*²), not (sin *u*)². A minus sign cannot start a bare input (write `sin(-u)`), and `min` and `max` always need brackets.

Powers go first, then × and ÷, then + and −, and a power binds to the right: `2^3^2` is 2^9, and `-u^2` is −(*u*²). Each expression can be up to 240 characters long (each end of a range counts on its own).



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/language.html#demo-reader

Why this number? The 240-character limit

OUR CHOICE Nothing in the math or the browser needs it.

A program lives in the toy's link, and a link holds up to 12 fields, so a limit keeps links short enough to paste into a message. The longest field of the toy's own programs is 53 characters. The limit was 120 until October 10, 2026, when it was raised to 240, for people who write longer formulas. A link written with the old limit loads the same as before.

If it were chosen differently. The link could carry much more (a text option may be up to 24,000 characters), so the limit is not what stops longer fields. Two other guards would be next: expressions nest at most 32 deep and have at most 400 pieces. The cost of longer fields is longer links and slower builds, because each expression is worked out for every point at every moment (at the default, 13 positions: the 12 moments and the closing target). At 240 characters a field can take up to twice as long to build as at 120, and a link with every field at the limit grows from about 1,400 to about 1,900 characters once compressed (`docs/lab/EQUATION-FIELDS.md`). The graph and surface plotters keep 120.

Where it lives: `FIELD_MAX` in `src/packs/splat-equation.js` (240); `MAX_LENGTH` (120, still the plotters' limit), `MAX_DEPTH` , and `MAX_NODES` in `src/equation.js` ; `TEXT_MAX` in `src/state.js` .

The grammar

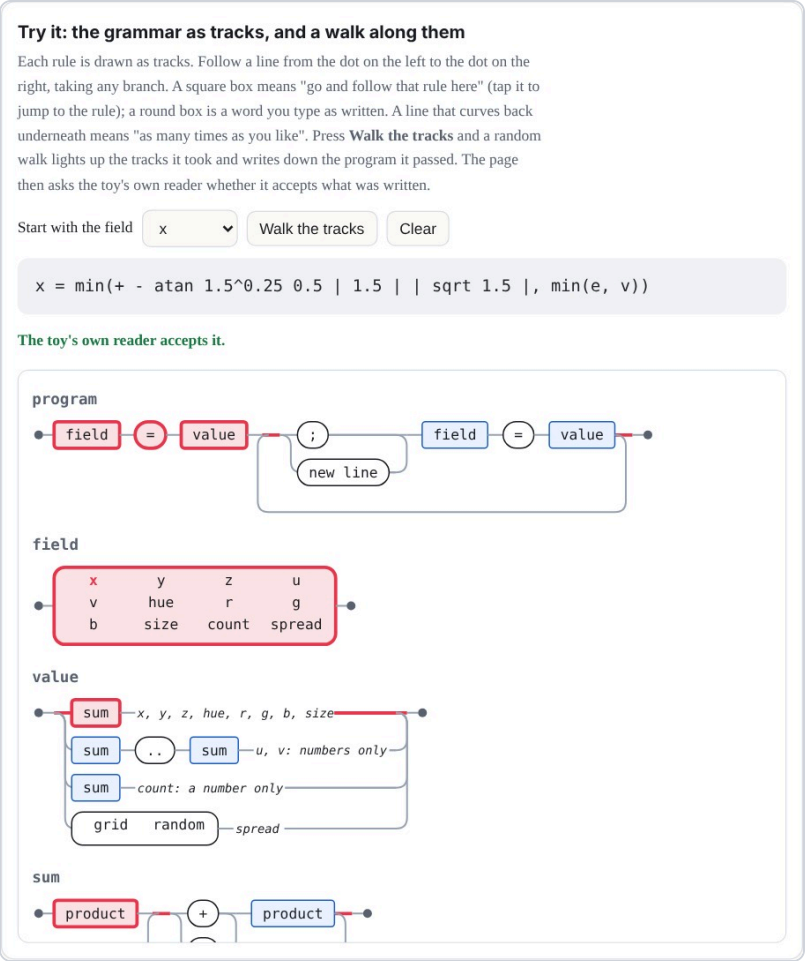
Here is the core grammar. Note: words in quotes must be typed exactly as written below.

A grammar is a short list of rules for what counts as a valid program. Its notation follows EBNF, a standard way to write such rules [8]:

- `=` means "is made of": the name on the left is built from the pattern on the right.
- `|` means "or". (`"|"` in quotes is the absolute-value sign you type.)
- `{ }` means "repeat what is inside any number of times, including none".
- `[ ]` means "optional": what is inside may be there or not.
- Quotes mark text you type exactly. Plain parentheses group choices together. Words with no quotes, such as `sum` , are names of other rules. A note in parentheses at the end of a line is only a comment.
- An **atom** is the smallest piece of an expression that can stand on its own: a number, a name, a function with its input, or something in brackets.

```
program    = field "=" value { ( ";" | new line ) field "=" value }
field      = "x" | "y" | "z" | "u" | "v" | "hue" | "r" | "g" | "b"
           | "size" | "count" | "spread"
value      = sum                          (x, y, z, hue, r, g, b, size)
           | sum ".." sum                 (u, v: numbers only, and no min or max)
           | sum                          (count: a number only)
           | "grid" | "random"            (spread)
sum        = product { ( "+" | "-" ) product }
product    = signed { ( "*" | "/" ) signed | power }   (nothing between: 2u)
signed     = { "-" | "+" } power
power      = atom [ "^" signed ]
atom       = number | "u" | "v" | "t" | "pi" | "e"
           | function "(" sum ")"
           | function bare                (sin 2u, sin u cos u, sin u^2)
           | ( "min" | "max" ) "(" sum "," sum { "," sum } ")"
           | bracket | "|" sum "|"        (not directly inside another | |)
bracket    = "(" sum ")" | "[" sum "]" | "{" sum "}"
```

```
bare       = simple { simple }           (stops at the next function or bracket)
simple     = ( number | "u" | "v" | "t" | "pi" | "e" ) [ "^" signed ]
function   = "sin" | "cos" | "tan" | "asin" | "acos" | "atan"
           | "arcsin" | "arccos" | "arctan" | "sinh" | "cosh" | "tanh"
           | "exp" | "log" | "ln" | "log10" | "sqrt" | "abs" | "floor"
                                         (one input; min and max are above)
```



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/language.html#demo-railroad

Time

t is time. A tap on the toy plays it from 0 to 2π in about 4 seconds, and the **t** slider in the Toy tab holds it anywhere in between. Anything with *t* in it moves: `u + t` turns a shape, `sin(t)` breathes, and `sin(3u − t)` sends a wave around.

Why this number? The two clocks

OUR CHOICE Both of them.

There are two clocks. The math clock, *t*, runs from 0 to 2π, one full circle, so that `sin` and `cos` of it come back exactly to where they started, and loops are seamless. The wall clock is real time: a play of *t* lasts about 4 seconds, whatever 0 to 2π means to the math. The speed is eased, so a play starts and ends still.

If it were chosen differently. A different range for *t* would change what a loop looks like (a range of 1 would make `sin(t)` stop mid-wave). A different play time would only change how fast it looks.

Where it lives: `PLAY_SECS` in `src/packs/splat-equation.js` .

Behind the scenes, the toy builds your shape at 12 moments, *t* = 0, π/6, 2π/6, and so on, and each copy slides every splat in a straight line to its place in the next copy (a morph, Chapter 2), so every splat is at its computed place at those 12 moments (up to the packing error described below) and slides in a straight line in between. A shape that turns dips very slightly inward between moments, about 3 percent. (With labs on, a **Moments of t** choice sets 13, 14, or 15 moments instead; 12 stays the default, so older links build as before.)

Why this number? Twelve moments

OUR CHOICE, NEAR A LIMIT Our choice; each moment also costs one of the 15 parts.

Twelve copies of 10,000 points (the most `count` allows) make 120,000 splats, the Low tier's largest budget, so even the weakest device can hold the biggest program. (The kit keeps 2 percent in reserve, so on that device each copy is cut to 9,800 points.) Each moment is its own part (Chapter 2), and 12 parts plus the body fit in the limit of 15.

If it were chosen differently. More moments make a turning shape rounder, because a splat slides in a straight line between moments and so dips inward in between. The worst dip is 1 − cos(π ÷ moments): 3.4 percent at 12 moments, 2.2 percent at 15, and 0.9 percent at 24. But the points per copy shrink on the weakest device: 9,800 at 12, 7,840 at 15, and 4,900 at 24. And because each moment is a part, 15 is the most without a bigger change. Color and size step at the moments rather than sliding, so more moments would smooth color changes too. The moments demo below lets you try every count. The labs choice **Moments of t** now offers 12, 13, 14, or 15; the Kit lab measured the worst dip of the splats at 3.41, 2.91, 2.51, and 2.19 percent (`docs/lab/MOMENTS.md`).

Where it lives: `KNOTS` (the default, 12) and `MOMENTS` (the choices) in `src/packs/splat-equation.js` ; the points per copy are floor(0.98 × budget ÷ moments).

Why is each moment a part, and what else could work?

NOT TESTED YET The alternatives below are ideas, not built.

The toy builds 12 complete copies of the shape, one for each moment of *t*. It shows one copy at a time and slides each into the next with the morph channel, so each copy needs its own part number, to be shown or hidden on its own. Other ways to do it:

- **Store the moments in a keyframe texture** and blend them in the shader. This needs no parts and allows more moments, but it is a change to the engine.
- **Compile the formulas to a shader**, so *t* is continuous and there are no copies at all. This works because every splat is a pure function of its own (*u*, *v*, *t*), with nothing shared between splats, so the graphics chip could work them all out at once. It is the larger experiment.
- **Recompute on the processor every frame.** This is too slow at 10,000 or more points, with a fresh sort every frame.

Why this number? Twelve bits

FORMAT LIMIT It follows from how much room each splat has.

What is stored for each splat's moment is the move from its place to its target place (the displacement), as three whole numbers, one per axis, each of 12 bits: 4,096 steps across a range of ±2 toy units. These are real binary digits, not a figure of speech. A toy is fitted to a sphere 1.9 units across, so any move fits, and one step is about 1/1,900 of the toy's width. Rounding to a step is the "tiny error" the manual mentions: at most half a step, as long as the move is inside the range (a move beyond ±2 units is cut off at the end of the range).

Each splat's animation data is four 32-bit numbers, and two of them are already taken (the part number with its flags, and the kind of motion). That leaves two. A 32-bit float holds any whole number up to 2^{24} exactly, and two 12-bit numbers make 24 bits, so one float carries two axes. The third axis and a channel number share the other.

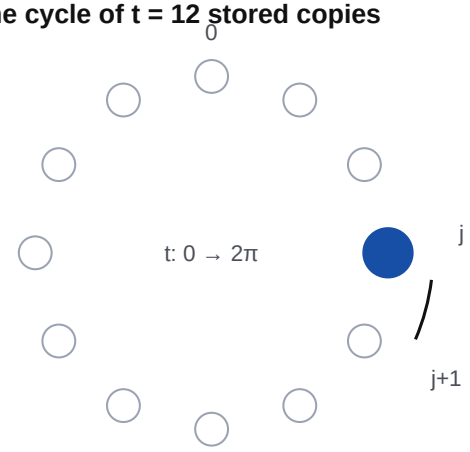
```
z = qx × 4096 + qy          (two 12-bit numbers, exact)
w = qz + 4096 × channel     (one 12-bit number and the channel)
```

If it were chosen differently. More bits would not fit two axes in one float, so the toy would need another float per splat (every float is already spoken for) or a texture. Fewer bits would make the steps visibly coarse.

Where it lives: `encodeMorphs` in `src/kit.js` ; `MORPH_RANGE` in `src/effects.js` .

Approximations in time. The shape is sampled at 12 moments (or 13 to 15, if you chose that in labs), not solved for every *t*. Only each splat's *position* slides toward the next copy, and that target is packed into 12 bits per axis over a fixed range, so tiny errors are possible. Its color, size, and rotation are built separately for each copy, so they can step at a copy boundary instead of blending. The "count" you type is the number of points sampled for each copy, not the total number of splats in the toy (Figure 3).

One cycle of t = 12 stored copies



Twelve copies, built at t = 0, 2π/12, 4π/12, …

Copy j shows and slides toward copy j + 1.

The other eleven are stored but hidden.

Where the splats go (the detail budget)

toy budget by device tier: Low 120,000 · Mid 240,000 · High 300,000 · Max 400,000

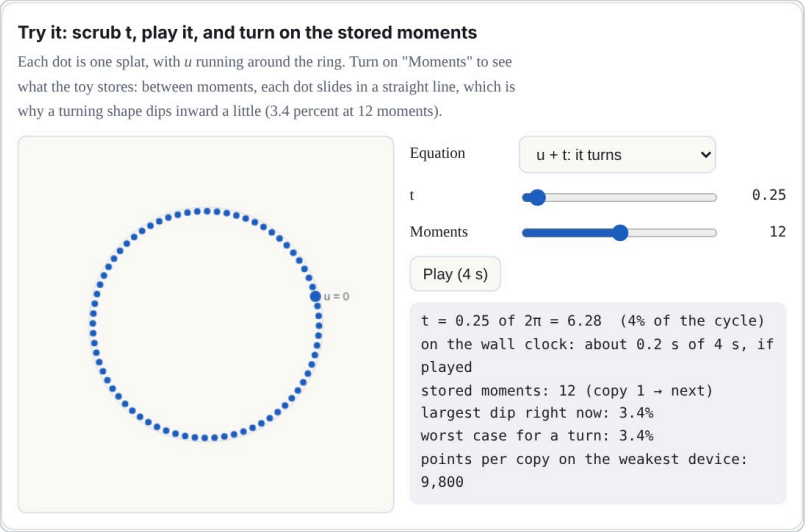
per copy: ⌊0.98 × toy budget ÷ 12⌋ (Low 9,800 … Max 32,666)

points per copy: the smaller of count and that budget

Fine: up to 4 × count, still within the budget

grid rounds down; points with no value are dropped

Figure 3. A program with *t* is stored as 12 copies, so the number of splats in memory is about 12 times the points in one copy. The device's tier sets the budget at the top; your count, the Fine setting, and the grid can only lower the points per copy.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/language.html#demo-moments

> **Make it loop.** Because *t* ends at 2π, a shape made of `sin(t)`, `cos(t)`, and angles like `u + t` ends exactly where it began. A shape with plain `t` in it (like `y = t`) jumps back at the end. `sin(t/2)` goes out and back once; `sin(2t)` twice.

A program without *t* stands still. Then a tap clears it and draws the splats again one by one, in the order they are made: the first value of *u* with every *v*, then the next *u*, and so on.

What the toy does for you

- **It fits the view.** The shape at its sampled moments of *t* is scaled to fit, so your units don't matter; only the sizes compared with the shape do.
- **It spaces the grid.** With `spread = grid`, the side of the shape that is longer gets more splats, so they sit about evenly apart. A shape that uses only *u* is a curve, and all the splats go along it.
- **It lights surfaces.** For a grid surface in Solid or Fine mode, each splat is a flat disc facing out from it (along the cross product of its slopes in *u* and *v*), lit from above. The **Light and shade** switch turns the lighting off, so you see your colors exactly. This is a simple shade, not physical lighting: your base color is multiplied by 0.62 + 0.48 × (how squarely the disc faces a fixed light) and a small highlight is added.
- **It skips impossible points.** Where an equation has no value (the square root of a negative number, a division by zero, the log of 0), that splat is left out. So is a point more than three times farther out than nearly all the others. A hue that can't be worked out (not a number, or infinite) becomes gray. A red, green, or blue that is not

a number becomes 0, and one that is too large or too small is clamped to 1 or 0 (+∞ is 1 and −∞ is 0). A size that can't be worked out becomes a small default. If a splat's next moment is impossible, it holds still instead of flying off.

- **It explains mistakes.** Each field that can't be read gets one plain message, like *“z: That can't be drawn: I don't know “q”.”*, and the shape on the stage stays as it was. Others are *“the number “1.2.” has two points.”* and *“w” is not a field. Use x, y, z, u, v, hue, r, g, b, size, count, spread.*

Why it is safe

The reader never runs what you type. It breaks the text into numbers, the letters *u*, *v*, and *t*, the names in the lists above and the signs, and builds a small tree of arithmetic from them. There is no way to name anything else, so the worst a program can do is draw a strange shape. Your program lives in the toy's link ("Copy link" in the Share tab), so you can send it to anyone.

CHAPTER 4

A gallery of programs

The first eight are the toy's own programs; the rest are small ideas to copy and change. Each button opens the toy with that program; then tap it to play *t*. While the toy is being tried out, it is on the shelf only when Splashery is opened with `?labs=1`, but these links always open it.

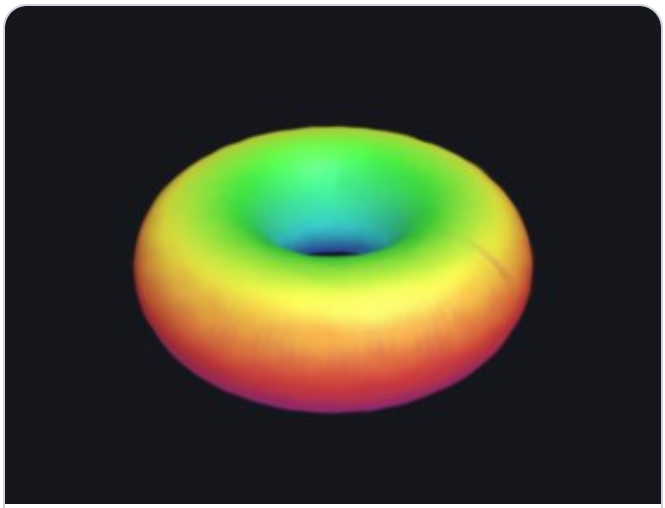


Sphere

A sphere from two angles, u around and v from pole to pole, spinning with u + t. Six stripes from floor(6u/(2pi)).

```
x = sin(v)cos(u + t)
y = cos(v)
z = sin(v)sin(u + t)
u = 0 .. 2pi
v = 0 .. pi
hue = floor(6u/(2pi))/6 + 0.04
size = 0.035
count = 7000
```

Open it in the toy



Torus

A tube around a ring. v + t rolls the tube around itself, so it turns inside out as you watch.

```
x = (2 + cos(v + t))cos(u)
y = sin(v + t)
z = (2 + cos(v + t))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = v/(2pi)
size = 0.09
count = 6000
```

Open it in the toy

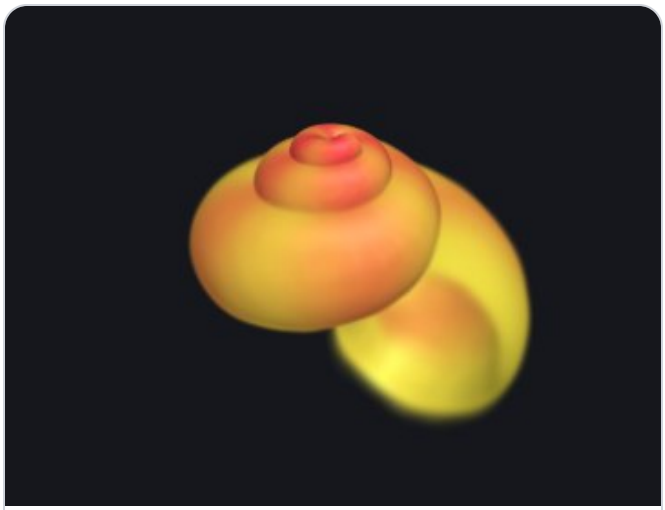


Möbius strip

A band with a half twist: cos(u/2) turns the band over once as u goes around. The colors flow along it.

```
x = (1 + v cos((u + t)/2))cos(u + t)
y = v sin((u + t)/2)
z = (1 + v cos((u + t)/2))sin(u + t)
u = 0 .. 2pi
v = -0.4 .. 0.4
hue = u/(2pi)
size = 0.035
count = 4000
```

Open it in the toy



Seashell

A tube that grows by exp(0.1u) as it winds three times around, turning with u + t. Its splats grow too.

```
x = exp(0.1u)cos(u + t)(1.2 + cos v)
y = exp(0.1u)(sin(v) - 2.4) + 3
z = exp(0.1u)sin(u + t)(1.2 + cos v)
u = 0 .. 6pi
v = 0 .. 2pi
hue = 0.02 + 0.1u/(6pi) +
0.03cos(4u)
size = 0.14exp(0.1u)
count = 10000
```

Open it in the toy

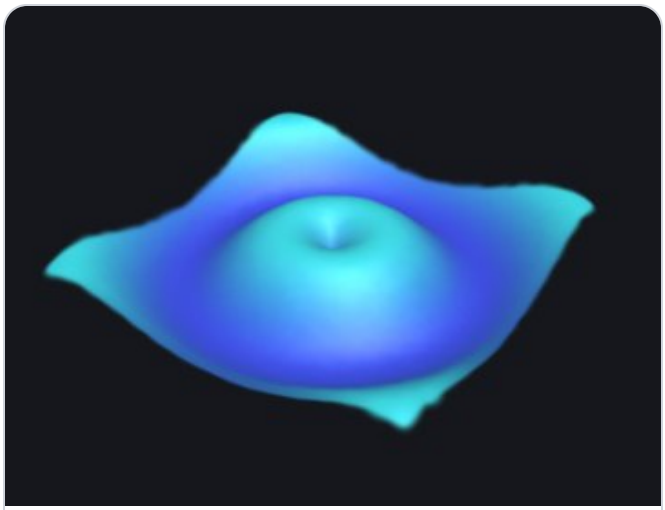


Trefoil knot

The simplest real knot, a curve in u alone, turned about the vertical by the rotation formulas in t.

```
x = (sin(u) + 2sin(2u))cos(t) -
sin(3u)sin(t)
y = cos(u) - 2cos(2u)
z = -(sin(u) + 2sin(2u))sin(t) -
sin(3u)cos(t)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.16
count = 2500
```

Open it in the toy

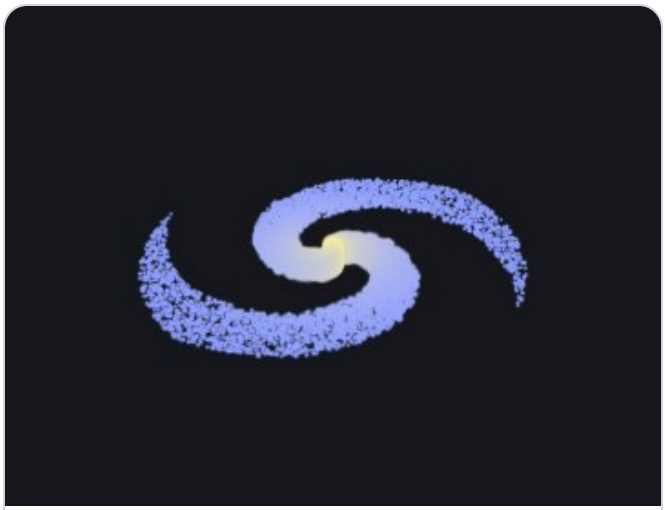


Wave

A flat sheet in u and v with ripples running out from the middle: sin(distance − t).

```
x = u
y = 0.45sin(2sqrt(u^2 + v^2) - t)
z = v
u = -3 .. 3
v = -3 .. 3
hue = 0.58 - 0.07sin(2sqrt(u^2 +
v^2) - t)
size = 0.07
count = 6000
```

Open it in the toy

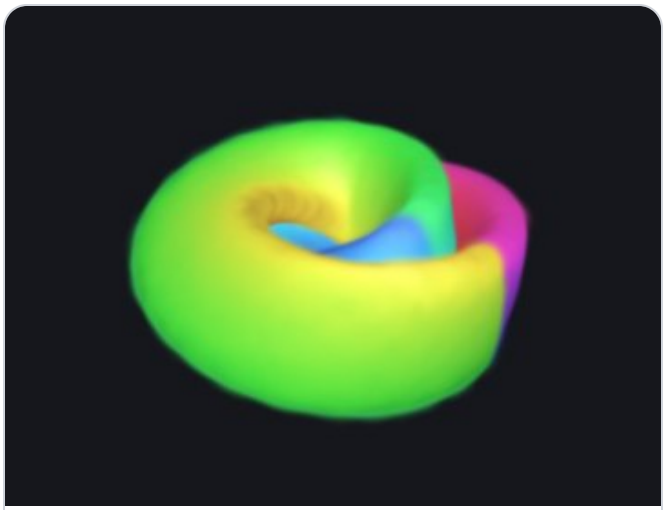


Spiral galaxy

Two arms from floor(v) (0 or 1, half a turn apart), scattered at random, turning once. Colors from r, g, and b.

```
x = u^1.5 cos(pi floor(v) + 5u +
1.2(v - floor(v)) - t)
y = 0.06(1 - u)sin(91v + 37u)
z = u^1.5 sin(pi floor(v) + 5u +
1.2(v - floor(v)) - t)
u = 0.02 .. 1
v = 0 .. 2
r = 0.55 + 0.45(1 - u)^2
g = 0.62 + 0.33(1 - u)^2
b = 1 - 0.45(1 - u)^3
size = 0.006 + 0.012(1 - u)^2
count = 10000
spread = random
```

Open it in the toy



Klein bottle (figure eight)

The figure-eight Klein bottle: a figure-eight tube around a ring, with a half twist, so it has no inside or outside.

```
x = (2 + cos(u/2)sin(v + t) -
sin(u/2)sin(2v + 2t))cos(u)
y = sin(u/2)sin(v + t) +
cos(u/2)sin(2v + 2t)
z = (2 + cos(u/2)sin(v + t) -
sin(u/2)sin(2v + 2t))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = v/(2pi)
size = 0.13
count = 8000
```

Open it in the toy



Hello, circle

The smallest program: one loop of u makes a ring of 200 splats. No v, no t.

```
x = cos(u)
y = sin(u)
z = 0
u = 0 .. 2pi
hue = u/(2pi)
size = 0.06
count = 200
```

Open it in the toy

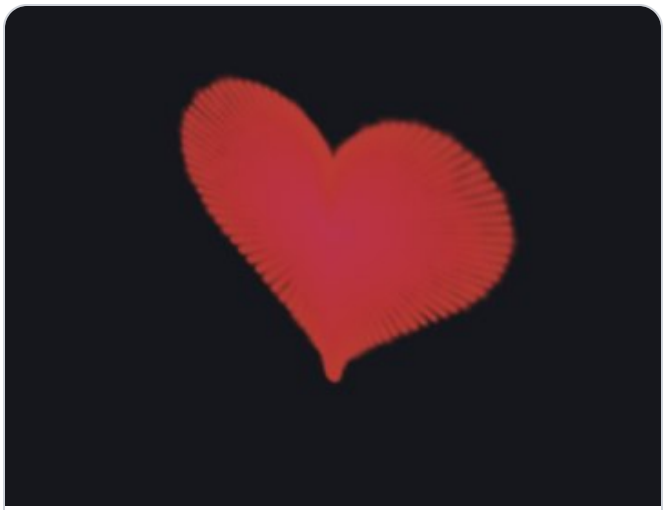


Bouncing spring

Three turns of a helix whose height breathes with sin(t).

```
x = cos(u)
y = (0.09 + 0.035sin(t))u - 0.9
z = sin(u)
u = 0 .. 6pi
hue = 0.08 + u/(6pi)/3
size = 0.07
count = 1500
```

Open it in the toy



Beating heart

A famous heart curve, filled in by v from the middle out, beating twice per cycle.

```
x = v(1 + 0.08sin(2t))16sin(u)^3
y = v(1 + 0.08sin(2t))(13cos(u) -
5cos(2u) - 2cos(3u) - cos(4u))
z = 0
u = 0 .. 2pi
v = 0 .. 1
hue = 0.97 + 0.04v
size = 0.35
count = 4000
```

Open it in the toy

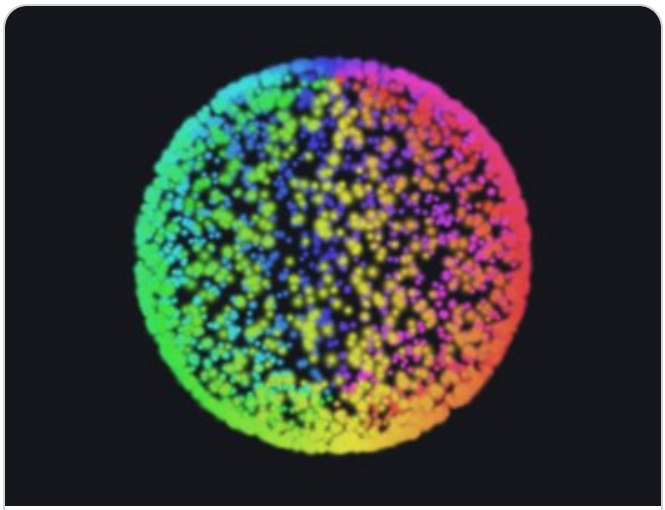


Rainbow ribbon

Color from r, g, and b: three cosines about a third of a turn apart make a rainbow along u.

```
x = u
y = 0.4sin(2u - t)
z = v
u = -3 .. 3
v = -0.4 .. 0.4
r = 0.5 + 0.5cos(u)
g = 0.5 + 0.5cos(u - 2)
b = 0.5 + 0.5cos(u + 2)
size = 0.05
count = 4000
```

Open it in the toy



Firework

Splats scattered at random over a sphere (v is the height, from −1 to 1) fly out, droop, and come back.

```
x = (0.2 + sin(t/2))sqrt(1 -
v^2)cos(u)
y = (0.2 + sin(t/2))v -
0.35sin(t/2)^2
z = (0.2 + sin(t/2))sqrt(1 -
v^2)sin(u)
u = 0 .. 2pi
v = -1 .. 1
hue = u/(2pi)
size = 0.018
count = 3000
spread = random
```

Open it in the toy

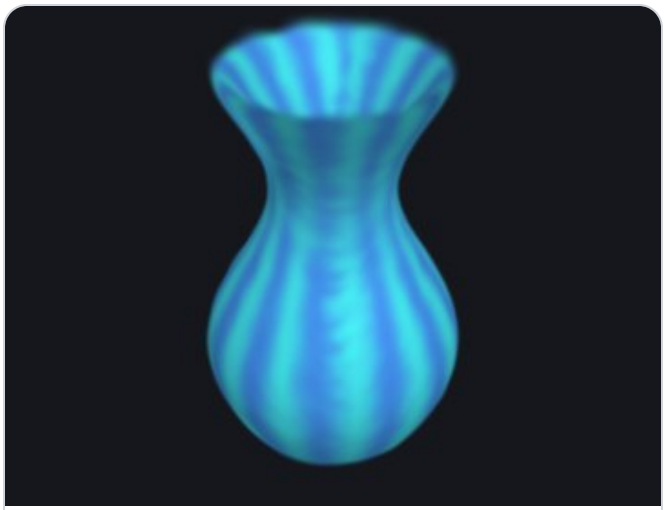


Opening flower

Five petals from cos(5u), and a cup that opens and closes with sin(t/2).

```
x = v(0.6 + 0.4cos(5u))cos(u)
y = (0.15 + 0.6sin(t/2))v^2
z = v(0.6 + 0.4cos(5u))sin(u)
u = 0 .. 2pi
v = 0 .. 1
hue = 0.92 - 0.12v
size = 0.035
count = 5000
```

Open it in the toy



Vase (no t)

A shape without t stands still, so a tap draws its splats again one by one, in the order they are made.

```
x = (0.5 + 0.2sin(3v))cos(u)
y = v
z = (0.5 + 0.2sin(3v))sin(u)
u = 0 .. 2pi
v = 0 .. 2.2
hue = 0.55 + 0.05sin(12u)
size = 0.045
count = 4000
```

Open it in the toy

Sixty more programs

A second gallery, in six groups of ten, written and then checked by hand: each program reads, builds, and looks like its title. A few are honest look-alikes or samples of a named system, and say so; none is a full solution of the system it names. Each card has the picture, one line about the shape, the program (folded), and a **Run it** link that opens the Splat equation toy with that program, in a new tab. That toy is a **labs** toy: it is not on the ordinary shelf, but a direct link like these opens it even with labs off, and its sharper kernel setting needs labs turned on. Every program fits the toy's own limits: 12 fields, 240 characters to an expression (each end of a range counts on its own; these programs are all under 120, so they would fit the old limit too), and at most 10,000 points.

Classic surfaces (10 programs)

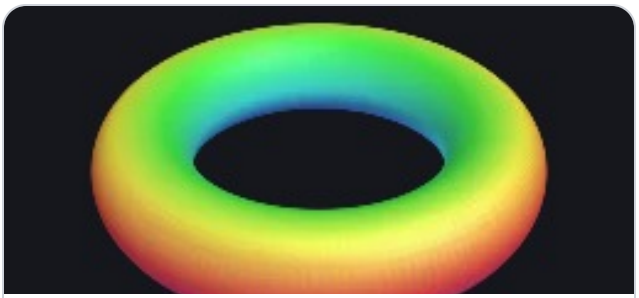


Spun sphere

A sphere turns once around its vertical axis, and its hue runs around the axis (by longitude), so the colors turn with it.

```
x = sin(v)cos(u+t)
y = cos(v)
z = sin(v)sin(u+t)
u = 0 .. 2pi
v = 0 .. pi
hue = u/(2pi)
size = 0.04
count = 6000
```

Run it (a labs toy) About the shape



Rainbow torus

A round tube wraps around a ring as its colors follow the small circle.

```
x = (2+0.55cos(v))cos(u)
y = 0.55sin(v)
z = (2+0.55cos(v))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = v/(2pi)
size = 0.05
count = 7000
```

Run it (a labs toy) About the shape



Möbius ribbon

A narrow band turns through one half twist before its edges meet.

```
x = (1+vcos(u/2))cos(u)
y = vsin(u/2)
z = (1+vcos(u/2))sin(u)
u = 0 .. 2pi
v = -0.32 .. 0.32
hue = u/(2pi)
size = 0.035
count = 5000
```

Run it (a labs toy) About the shape

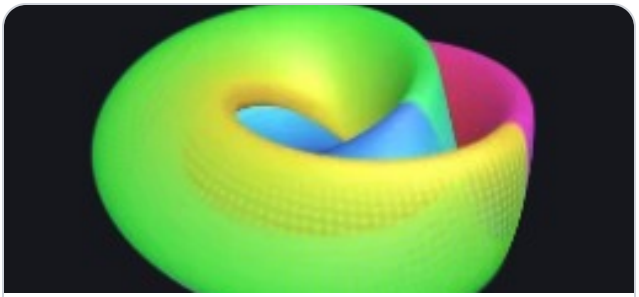
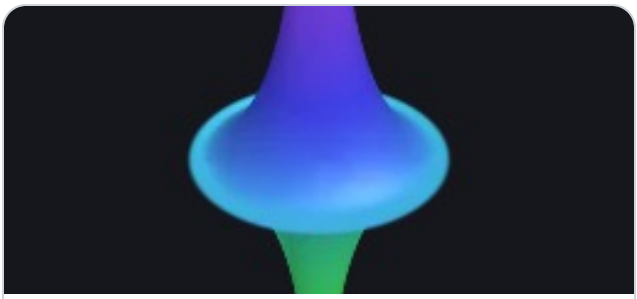


Figure-eight Klein bottle

This immersed Klein bottle is a figure-eight tube that crosses itself along a ring.

```
x = (2+cos(u/2)sin(v)-sin(u/2)sin(2v))cos(u)
y = sin(u/2)sin(v)+cos(u/2)sin(2v)
z = (2+cos(u/2)sin(v)-sin(u/2)sin(2v))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = v/(2pi)
size = 0.07
count = 10000
```

Run it (a labs toy) About the shape

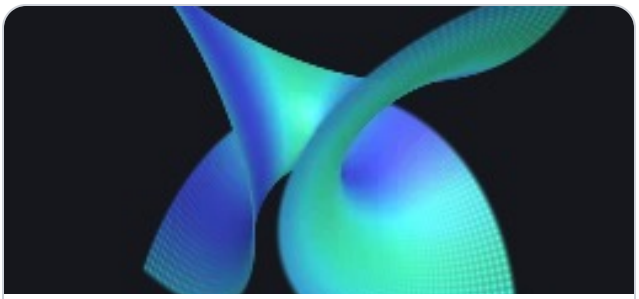


Pseudosphere

Two tractrix horns meet at a sharp rim: a surface that curves the opposite way from a sphere.

```
x = cos(u)/cosh(v)
y = 1.2(v-tanh(v))
z = sin(u)/cosh(v)
u = 0 .. 2pi
v = -2.6 .. 2.6
hue = 0.55+0.2v/2.6
size = 0.05
count = 10000
```

Run it (a labs toy) About the shape

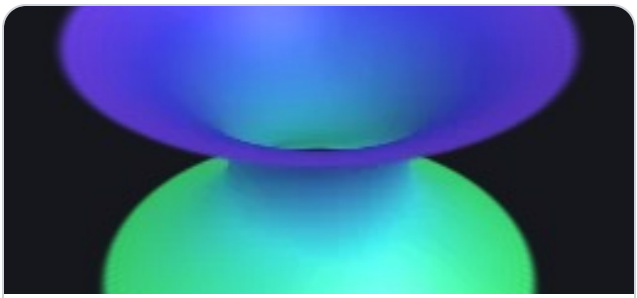


Enneper surface

A simple polynomial surface curls into spreading lobes.

```
x = u-u^3/3+uv^2
y = v-v^3/3+vu^2
z = u^2-v^2
u = -1.6 .. 1.6
v = -1.6 .. 1.6
hue = 0.55+0.1sin(4u)
size = 0.07
count = 10000
```

Run it (a labs toy) About the shape

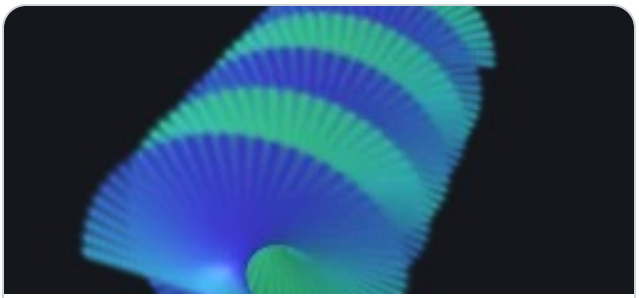


Catenoid

A catenoid pinches between two wide circular rims.

```
x = 0.45cosh(v)cos(u)
y = 0.45v
z = 0.45cosh(v)sin(u)
u = 0 .. 2pi
v = -1.6 .. 1.6
hue = 0.55+0.1v
size = 0.035
count = 7000
```

Run it (a labs toy) About the shape



Helicoid

A finite ruled helicoid patch winds like a ramp around the Z axis, with a helical edge on its outer rim.

```
x = vcos(u)
y = vsin(u)
z = 0.18u
u = -3pi .. 3pi
v = -1 .. 1
hue = 0.55+0.12sin(u)
size = 0.035
count = 7000
```

Run it (a labs toy) About the shape



Dini surface

Dini's surface twists a trumpet shape into a tight spiral.

```
x = 0.65cos(u)sin(v)
y = 0.65sin(u)sin(v)
z = 0.65(cos(v)+log(tan(v/2)))+0.12u
u = 0 .. 4pi
v = 0.18 .. 2.95
hue = u/(4pi)
size = 0.03
count = 8000
```

Run it (a labs toy) About the shape



Shell growth

A logarithmic tube grows wider as it winds away from its narrow tip.

```
x = exp(0.12u)(1+0.35cos(v))cos(u)
y = exp(0.12u)(1+0.35cos(v))sin(u)
z = exp(0.12u)(0.35sin(v))
u = 0 .. 6pi
v = 0 .. 2pi
hue = u/(6pi)
size = 0.04+0.01u
count = 10000
```

Run it (a labs toy) About the shape

Curves as tubes or beads (10 programs)



Lissajous curve

Four frequencies (2, 3, 4, and 7) trace a closed space curve that crosses itself, with a fine bead tube.

```
x = sin(3u+t)+0.16cos(7u)
y = 0.12cos(2u)
z = cos(4u+t)+0.16sin(7u)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.045
count = 3200
```

Run it (a labs toy) About the shape

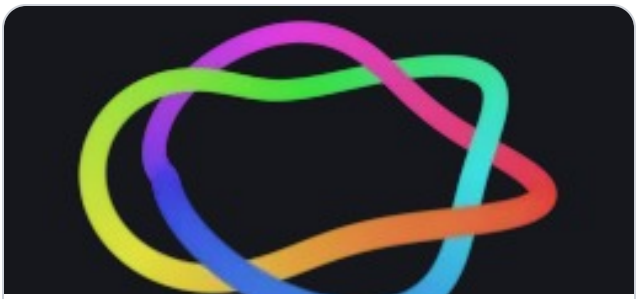


Trefoil tube

A tube follows the classic three-crossing trefoil knot and rotates in place.

```
x = (sin(u)+2sin(2u))cos(t)-sin(3u)sin(t)
y = cos(u)-2cos(2u)
z = -(sin(u)+2sin(2u))sin(t)-sin(3u)cos(t)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.08
count = 3000
```

Run it (a labs toy) About the shape



Cinquefoil knot

Five over-and-under crossings wind around a torus in this thicker knot.

```
x = (2+0.35cos(5u))cos(2u+t)
y = 0.35sin(5u)
z = (2+0.35cos(5u))sin(2u+t)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.055
count = 3500
```

Run it (a labs toy) About the shape



Rose curve beads

The curve cos(6u) traced in a plane makes a rose of 12 petals (an even multiple makes twice as many), and a gentle vertical wave lifts it into a dotted three-dimensional curve.

```
x = cos(6u)cos(u)
y = 0.15sin(3u)
z = cos(6u)sin(u)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.04
count = 3000
```

Run it (a labs toy) About the shape

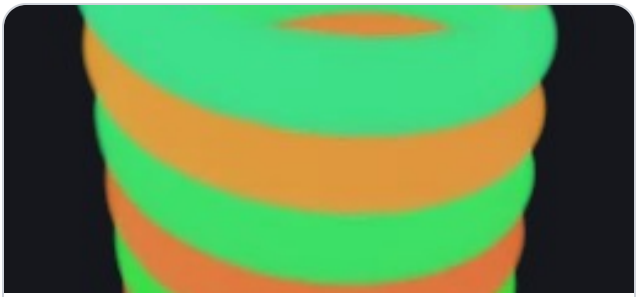


Trefoil knot beads

This closed string of beads traces a trefoil, the simplest knot (a torus knot that wraps three times one way and twice the other).

```
x = (2+cos(2u))cos(3u+t)
y = sin(2u)
z = (2+cos(2u))sin(3u+t)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.05
count = 3500
```

Run it (a labs toy) About the shape



Double helix

Two spiral strands wind together around a long axis.

```
x = 1.1cos(3u+pi floor(2v))
y = 1.4(u/pi-1)
z = 1.1sin(3u+pi floor(2v))
u = 0 .. 2pi
v = 0 .. 1
hue = 0.02+0.3floor(2v)+0.1u/(2pi)
size = 0.09
count = 10000
spread = random
```

Run it (a labs toy) About the shape



Slinky coil

A spring winds upward in eight turns as a narrow ribbon.

```
x = (1+0.08v)cos(u)
y = 0.1u+0.08vcos(u/8)
z = (1+0.08v)sin(u)
u = 0 .. 16pi
v = -1 .. 1
hue = u/(16pi)
size = 0.035
count = 5000
```

Run it (a labs toy) About the shape

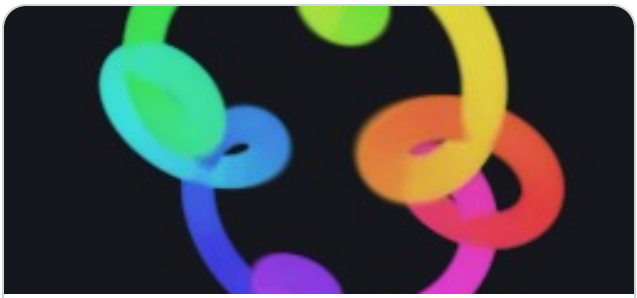


Lorenz-like look-alike

This butterfly loop is a Lorenz-like look-alike, not a Lorenz-system solution.

```
x = 2.4sin(u)
y = 1.5sin(2u)+0.25sin(6u)
z = 1.1cos(u)+0.3cos(3u)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.07
count = 4000
```

Run it (a labs toy)



Rössler-like look-alike

A looping spiral suggests a Rössler attractor but is only a parametric look-alike.

```
x = cos(u)+0.35cos(7u)
y = sin(u)+0.35sin(7u)
z = 0.45sin(3u)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.045
count = 3000
```

Run it (a labs toy)



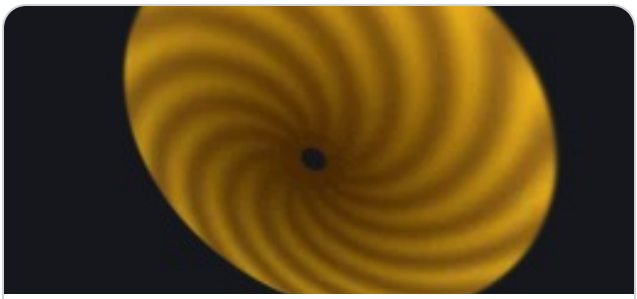
Five-petal flower

Five broad petals rise from a shallow cup around a small center.

```
x = v(0.55+0.45cos(5u))cos(u)
y = 0.5v^2+0.1cos(5u)
z = v(0.55+0.45cos(5u))sin(u)
u = 0 .. 2pi
v = 0 .. 1
hue = 0.9-0.12v
size = 0.035
count = 6000
```

Run it (a labs toy) About the shape

Nature-like shapes (10 programs)

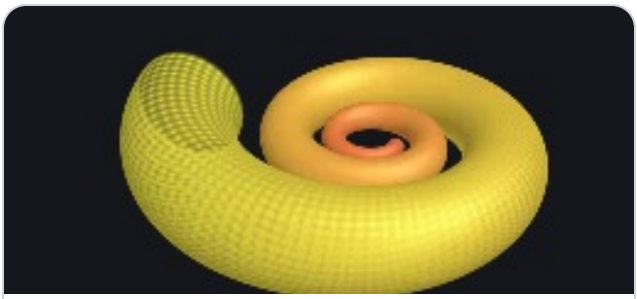


Sunflower-style spiral

A face-on disk shades from dark seeds to a golden rim, with Fibonacci-style spiral arms drawn in the color.

```
x = 1.8sqrt(u)cos(2pi v)
y = 1.8sqrt(u)sin(2pi v)
z = 0.25(1-u)
u = 0.002 .. 1
v = 0 .. 1
r = 0.3+0.7sqrt(u)(0.75+0.25sin(26pi v-24sqrt(u)))
g = 0.15+0.6sqrt(u)(0.75+0.25sin(26pi v-24sqrt(u)))
b = 0.02+0.1u
size = 0.04
count = 10000
```

Run it (a labs toy) About the shape

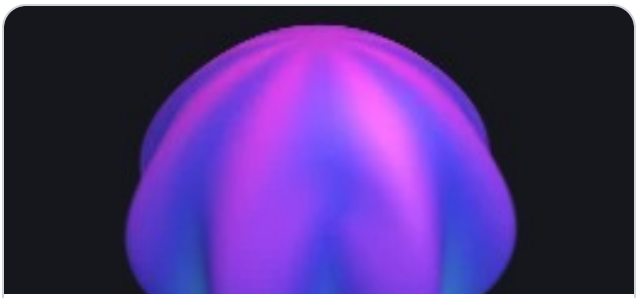


Nautilus coil

A widening spiral tube evokes a nautilus shell with a visible open center.

```
x = exp(0.13u)(1+0.3cos(v))cos(u)
y = 0.3exp(0.13u)sin(v)
z = exp(0.13u)(1+0.3cos(v))sin(u)
u = 0 .. 5pi
v = 0 .. 2pi
hue = 0.04+0.12u/(5pi)
size = 0.035+0.008u
count = 10000
```

Run it (a labs toy) About the shape

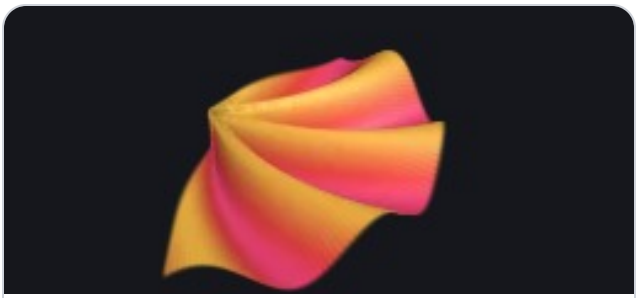


Jellyfish bell

A dome-like bell pulses as its rim widens and narrows once each cycle.

```
x = (0.9+0.12sin(t))sin(v)
(1+0.1cos(8u)sin(v)^6)cos(u)
y = 0.9cos(v)-0.2
z = (0.9+0.12sin(t))sin(v)
(1+0.1cos(8u)sin(v)^6)sin(u)
u = 0 .. 2pi
v = 0 .. 0.6pi
hue = 0.82+0.05cos(8u)-0.1v
size = 0.05
count = 10000
```

Run it (a labs toy) About the shape

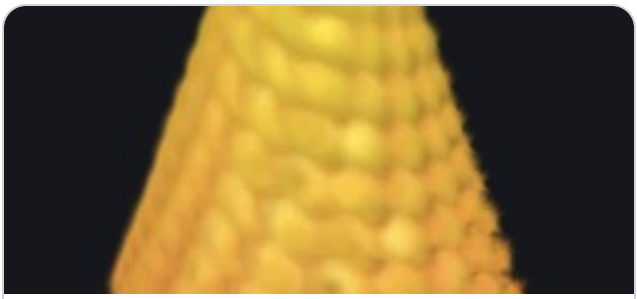


Coral fan

Branching sine ribs spread across a fan-shaped sheet.

```
x = vcos(u)
y = 0.16sin(9u+2v)
z = vsin(u)
u = -1.3 .. 1.3
v = 0 .. 1.8
hue = 0.03+0.08sin(9u+2v)
size = 0.025
count = 7000
```

Run it (a labs toy) About the shape

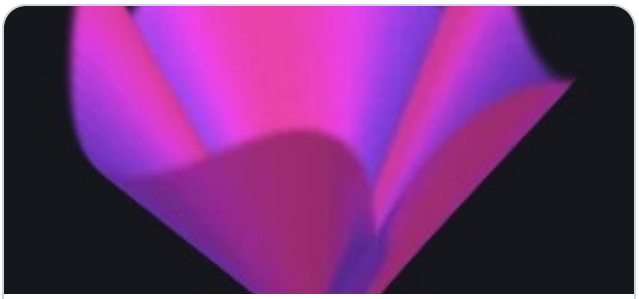


Pinecone spiral

Scales follow a Fibonacci angle and taper toward the top of the cone.

```
x = (0.9-0.65v)(1+0.09sin(16pi u+25v)sin(26pi u-25v))cos(2pi u+2.4v)
y = 1.8v
z = (0.9-0.65v)(1+0.09sin(16pi u+25v)sin(26pi u-25v))sin(2pi u+2.4v)
u = 0 .. 1
v = 0 .. 1
hue = 0.07+0.05v+0.025sin(16pi u+25v)sin(26pi u-25v)
size = 0.045
count = 10000
```

Run it (a labs toy) About the shape



Sea anemone

Curled arms rise from a low base like a ring of waving tentacles.

```
x = (0.25+0.55v)cos(u+0.3sin(3u))
y = v(1+0.18sin(5u))
z = (0.25+0.55v)sin(u+0.3sin(3u))
u = 0 .. 2pi
v = 0 .. 1
hue = 0.82+0.08sin(5u)
size = 0.03
count = 6000
```

Run it (a labs toy) About the shape



Fern frond

A central stem and repeated leaflets make a simple fern-like frond.

```
x = u
y = 0.08sin(7piu)
z = v(0.15+0.65abs(sin(7piu)))
u = -1 .. 1
v = 0 .. 1
hue = 0.28+0.12v
size = 0.025
count = 6500
```

Run it (a labs toy) About the shape

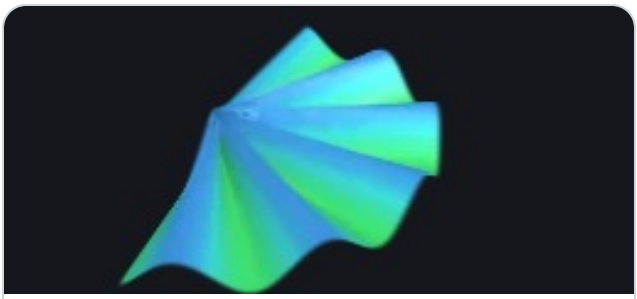


Dahlia bloom

Layered radial petals alternate in height to form a dense dahlia-like flower.

```
x = v(0.4+0.6abs(cos(12u)))cos(u)
y = 0.25sin(12u)*v
z = v(0.4+0.6abs(cos(12u)))sin(u)
u = 0 .. 2pi
v = 0 .. 1
hue = 0.88+0.1cos(12u)
size = 0.025
count = 8000
```

Run it (a labs toy) About the shape

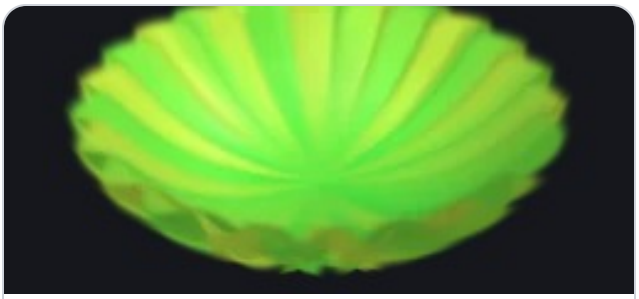


Sea fan branches

A flat branching fan uses nested waves to suggest coral growth.

```
x = vcos(u)
y = 0.12sin(11u)
z = vsin(u)
u = -1.5 .. 1.5
v = 0 .. 1.5
hue = 0.48+0.1sin(11u)
size = 0.028
count = 6500
```

Run it (a labs toy) About the shape



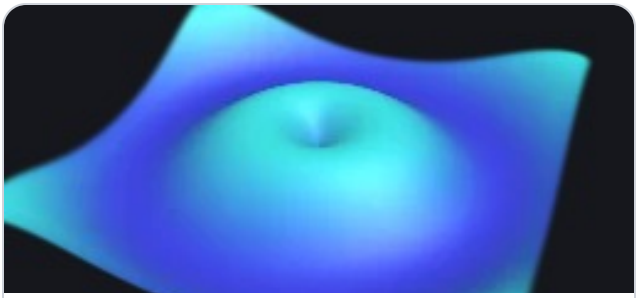
Lily pad

A scalloped circular leaf floats on a softly raised surface.

```
x = v(1+0.04cos(24u))cos(u)
y = 0.3v^3+0.12v
z = v(1+0.04cos(24u))sin(u)
u = 0.3 .. 2pi
v = 0 .. 1.2
hue = 0.3+0.04cos(14u)-0.05v
size = 0.06
count = 10000
```

Run it (a labs toy) About the shape

Things in motion (10 programs)

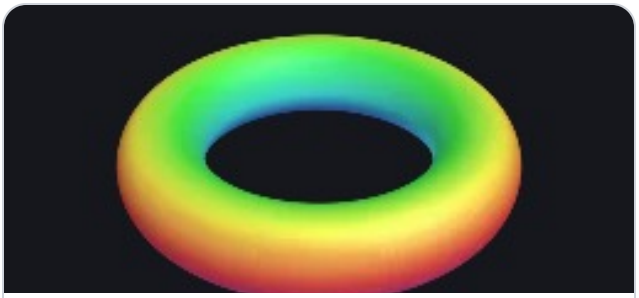


Traveling wave

Concentric ripples travel across a square sheet without breaking at the cycle seam.

```
x = u
y = 0.38sin(2sqrt(u^2+v^2)-t)
z = v
u = -2.8 .. 2.8
v = -2.8 .. 2.8
hue = 0.58-0.08sin(2sqrt(u^2+v^2)-t)
size = 0.09
count = 10000
```

Run it (a labs toy) About the shape

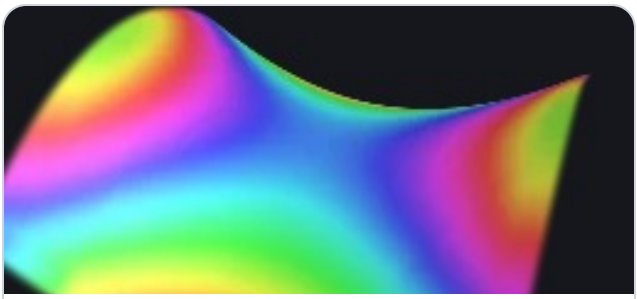


Breathing torus

The major and minor radii expand together and return once per cycle.

```
x = (1.7+0.45cos(v))(1+0.12sin(t))cos(u)
y = 0.45(1+0.12sin(t))sin(v)
z = (1.7+0.45cos(v))(1+0.12sin(t))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = v/(2pi)
size = 0.045
count = 6500
```

Run it (a labs toy) About the shape



Rotating saddle

A saddle ripple rotates smoothly over a square patch.

```
x = u
y = 0.22(u^2-v^2)cos(t)+0.2uvsin(t)
z = v
u = -1.6 .. 1.6
v = -1.6 .. 1.6
hue = 0.6+1.2(0.22(u^2-v^2)cos(t)+0.2uvsin(t))
size = 0.06
count = 10000
```

Run it (a labs toy) About the shape

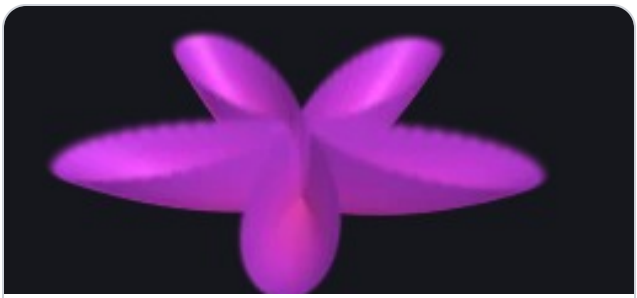


Orbit with a fast wave

One loop circles the center while a smaller wave on it, with three times as many crests around the loop, turns the other way.

```
x = cos(u+t)+0.35cos(3u-t)
y = 0.35sin(3u-t)
z = sin(u+t)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.045
count = 4500
```

Run it (a labs toy) About the shape

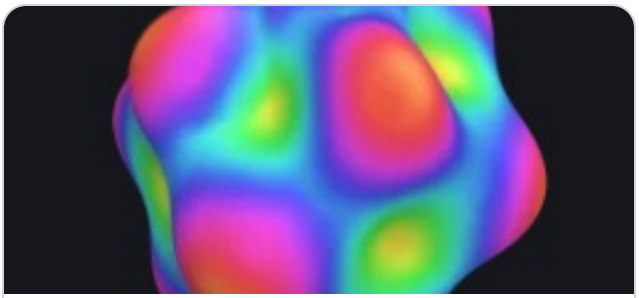


Opening flower

Petals lift and flatten in a smooth repeating bloom.

```
x = v(0.55+0.45cos(5u))cos(u)
y = (0.12+0.45(1+sin(t))/2)v^2
z = v(0.55+0.45cos(5u))sin(u)
u = 0 .. 2pi
v = 0 .. 1
hue = 0.9-0.1v
size = 0.03
count = 6000
```

Run it (a labs toy) About the shape

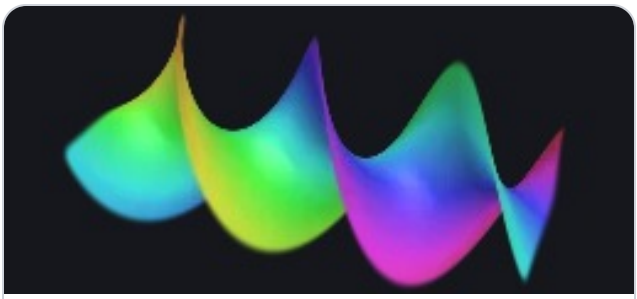


Breathing bumpy sphere

Four-fold bumps grow and shrink across a round surface as its color follows each bump.

```
x = sin(v)cos(u)(1+0.2cos(4u)cos(4v)sin(t+1.2))
y = cos(v)(1+0.2cos(4u)cos(4v)sin(t+1.2))
z = sin(v)sin(u)(1+0.2cos(4u)cos(4v)sin(t+1.2))
u = 0 .. 2pi
v = 0 .. pi
hue = 0.6+0.5cos(4u)cos(4v)sin(t+1.2)
size = 0.05
count = 10000
```

Run it (a labs toy) About the shape

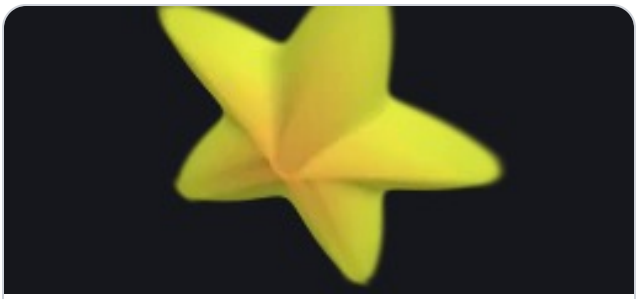


Twisting ribbon

A narrow ribbon rotates its cross-section through one full turn each cycle.

```
x = u
y = 1.1vcos(2u+t)
z = 1.1vsin(2u+t)
u = -2.4 .. 2.4
v = -1 .. 1
hue = 0.5+0.25v+0.1u
size = 0.06
count = 10000
```

Run it (a labs toy) About the shape



Pulsing star

Five radial points breathe in and out with the beat of a sine wave.

```
x = 1.8v(0.65+0.16sin(t))(0.45+0.55(1-
abs(sin(2.5u))))cos(u)
y = 1.8v(0.65+0.16sin(t))(0.45+0.55(1-
abs(sin(2.5u))))sin(u)
z = 0.3(1-v)
u = 0 .. 2pi
v = 0 .. 1
hue = 0.1+0.08v+0.04sin(t)
size = 0.05
count = 10000
```

Run it (a labs toy) About the shape

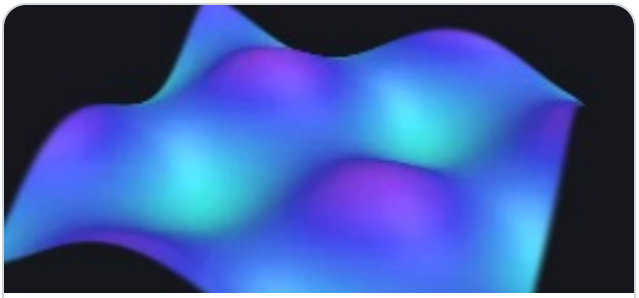


Wobbling helix

A helix sways side to side and returns to its starting pose each cycle.

```
x = cos(u+t)
y = 0.12u
z = sin(u+t)+0.2sin(t)
u = 0 .. 8pi
v = 0 .. 1
hue = u/(8pi)
size = 0.04
count = 5000
```

Run it (a labs toy) About the shape



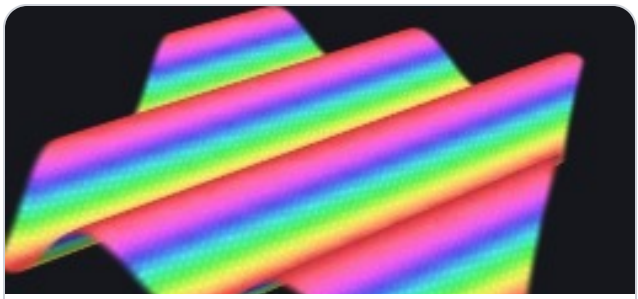
Standing wave

A rectangular sheet ripples in place with a standing two-axis wave.

```
x = u
y = 0.35sin(2u)sin(2v)cos(t)
z = v
u = -2 .. 2
v = -2 .. 2
hue = 0.62+0.12sin(2u)sin(2v)
size = 0.065
count = 10000
```

Run it (a labs toy) About the shape

Color play (10 programs)



Height hue bands

A ridged surface assigns a full rainbow cycle to its height.

```
x = u
y = 0.3sin(4u+3v)
z = v
u = -2 .. 2
v = -2 .. 2
hue = (sin(4u+3v)+1)/2
size = 0.04
count = 6000
```

Run it (a labs toy) About the shape

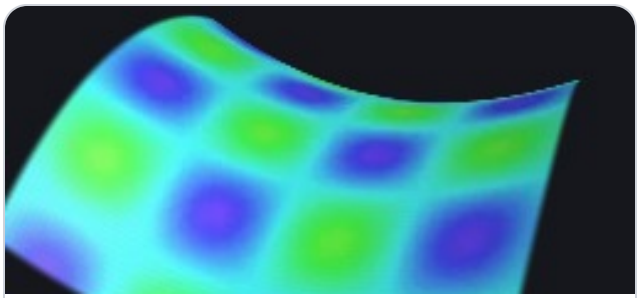


Angle around the torus

Hue runs once around the large ring, so each slice across the tube has one color.

```
x = (1.5+0.4cos(v))cos(u)
y = 0.4sin(v)
z = (1.5+0.4cos(v))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = u/(2pi)
size = 0.04
count = 6500
```

Run it (a labs toy) About the shape

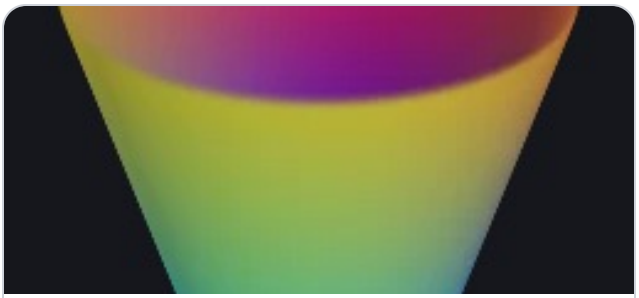


Curvature-like bands

Nested sine ridges color a saddle with a pattern that suggests changing curvature.

```
x = u
y = 0.15(u^2-v^2)
z = v
u = -2 .. 2
v = -2 .. 2
hue = 0.5+0.2sin(3u)cos(3v)
size = 0.035
count = 6500
```

Run it (a labs toy) About the shape



RGB sunset gradient

Red rises with height, green follows the horizontal angle, and blue fills the valleys.

```
x = cos(u)(1+0.2v)
y = v
z = sin(u)(1+0.2v)
u = 0 .. 2pi
v = -1 .. 1
r = 0.55+0.35v
g = 0.5+0.4sin(u)
b = 0.55-0.3v
size = 0.04
count = 6000
```

Run it (a labs toy) About the shape



Rainbow shell

A growing shell carries a rainbow gradient from its narrow tip to its wide lip.

```
x = exp(0.1u)cos(u)(1+0.25cos(v))
y = exp(0.1u)0.25sin(v)
z = exp(0.1u)sin(u)(1+0.25cos(v))
u = 0 .. 6pi
v = 0 .. 2pi
hue = u/(6pi)
size = 0.04+0.01u
count = 10000
```

Run it (a labs toy) About the shape

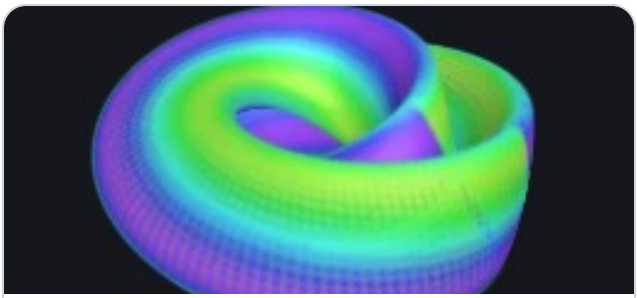


Blue-green sphere

An RGB gradient moves from deep blue at the bottom to green at the top.

```
x = sin(v)cos(u)
y = cos(v)
z = sin(v)sin(u)
u = 0 .. 2pi
v = 0 .. pi
r = 0.05
g = 0.1+0.85(1+cos(v))/2
b = 0.15+0.8(1-cos(v))/2
size = 0.05
count = 10000
```

Run it (a labs toy) About the shape

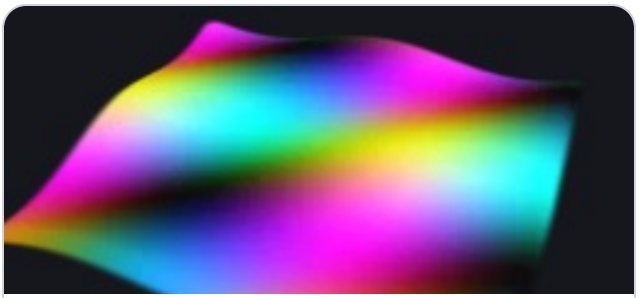


Klein bottle color flow

Alternating bands run around the immersed bottle and make its crossing easy to see.

```
x = (2+cos(u/2)sin(v)-sin(u/2)sin(2v))cos(u)
y = sin(u/2)sin(v)+cos(u/2)sin(2v)
z = (2+cos(u/2)sin(v)-sin(u/2)sin(2v))sin(u)
u = 0 .. 2pi
v = 0 .. 2pi
hue = 0.5+0.25sin(4v)
size = 0.07
count = 10000
```

Run it (a labs toy) About the shape



Plasma RGB sheet

Three offset waves make a smooth red, green, and blue color field.

```
x = u
y = 0.15sin(1.5u+1.5v)
z = v
u = -3 .. 3
v = -3 .. 3
r = 0.5+0.45sin(1.5u)
g = 0.5+0.45sin(1.5v+2)
b = 0.5+0.45sin(1.5u+1.5v+4)
size = 0.09
count = 10000
```

Run it (a labs toy) About the shape

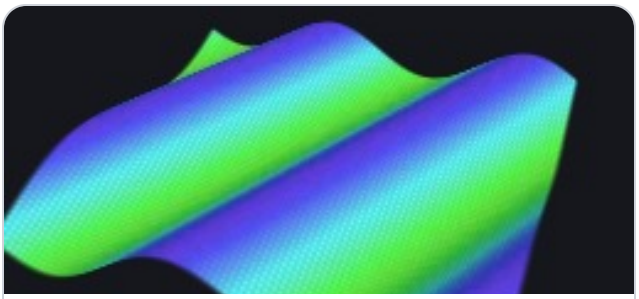


Longitude stripes

A sphere is divided into twelve clean color wedges by its longitude.

```
x = sin(v)cos(u)
y = cos(v)
z = sin(v)sin(u)
u = 0 .. 2pi
v = 0 .. pi
hue = floor(12u/(2pi))/12
size = 0.035
count = 7000
```

Run it (a labs toy) About the shape



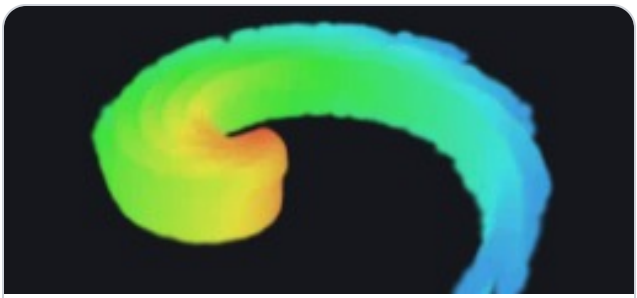
Rainbow wave

A moving ripple shifts both the sheet height and its rainbow colors.

```
x = u
y = 0.35sin(2u+v-t)
z = v
u = -3 .. 3
v = -3 .. 3
hue = 0.5+0.2sin(2u+v-t)
size = 0.04
count = 7000
```

Run it (a labs toy) About the shape

Math showpieces (10 programs)

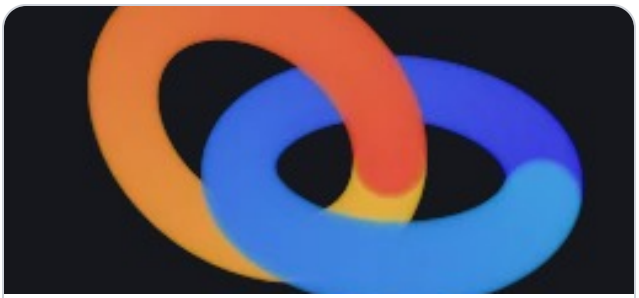


Thermal spiral

Warm colors gather near the core and cool colors trail along the spiral arms.

```
x = u cos(v+5u)
y = 0.12sin(13v+4u)
z = u sin(v+5u)
u = 0.02 .. 1
v = 0 .. 2
hue = 0.02+0.55u
size = 0.018
count = 9000
spread = random
```

Run it (a labs toy) About the shape



Hopf link

Two circles pass through each other once, the simplest link, drawn as two dotted rings in different colors.

```
x = 1.6(1-floor(2v))cos(u)+floor(2v)
(1.6cos(u)+1.6)-0.8
y = 1.6(1-floor(2v))sin(u)
z = 1.6floor(2v)sin(u)
u = 0 .. 2pi
v = 0 .. 1
hue = 0.02+0.55floor(2v)+0.1u/(2pi)
size = 0.14
count = 10000
spread = random
```

Run it (a labs toy) About the shape



Fourier epicycle curve

Three rotating harmonics add into one closed curve with a small bead radius.

```
x = cos(u)+0.45cos(3u)+0.2cos(5u)
y = 0.18sin(7u)
z = sin(u)+0.45sin(3u)+0.2sin(5u)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.04
count = 3500
```

Run it (a labs toy) About the shape

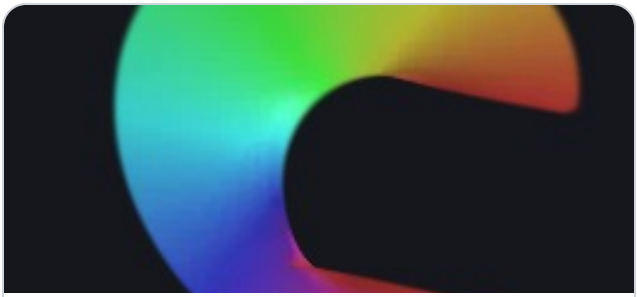


Superquadric

Fourth-power rounding turns a sphere into a soft box with six flattened faces.

```
x =
1.6cos(u)cos(v)/((cos(u)^4+sin(u)^4)^0.25(cos(v)^4+s
in(v)^4)^0.25)
y = 1.6sin(v)/(cos(v)^4+sin(v)^4)^0.25
z =
1.6sin(u)cos(v)/((cos(u)^4+sin(u)^4)^0.25(cos(v)^4+s
in(v)^4)^0.25)
u = 0 .. 2pi
v = -pi/2 .. pi/2
hue = 0.5+0.3v/pi+0.1sin(u)
size = 0.06
count = 10000
```

Run it (a labs toy) About the shape

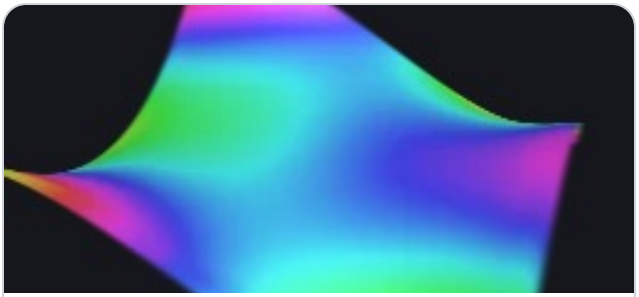


Riemann branch look

A square-root branch surface spirals through two sheets around its branch point.

```
x = sqrt(u)cos(v/2)
y = sqrt(u)sin(v/2)
z = 0.12v
u = 0.03 .. 1
v = 0 .. 4pi
hue = v/(4pi)
size = 0.03
count = 7000
```

Run it (a labs toy) About the shape

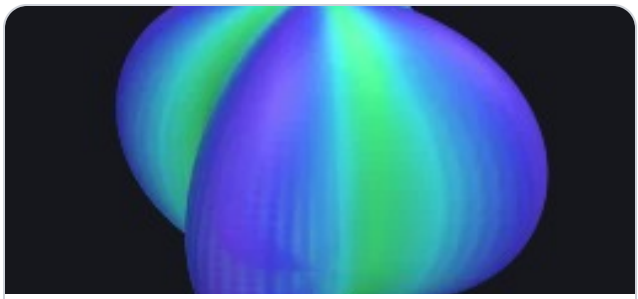


Monkey saddle

A cubic height field has three valleys meeting at one central saddle point.

```
x = u
y = 0.12(u^3-3uv^2)
z = v
u = -1.2 .. 1.2
v = -1.2 .. 1.2
hue = 0.55+0.1sin(3u)+0.2(u^3-3uv^2)
size = 0.05
count = 10000
```

Run it (a labs toy) About the shape



Spherical harmonic lobes

A spherical-harmonic-like radial formula gives the sphere four raised and lowered lobes.

```
x = (1+0.3sin(v)^2cos(4u))sin(v)cos(u)
y = (1+0.3sin(v)^2cos(4u))cos(v)
z = (1+0.3sin(v)^2cos(4u))sin(v)sin(u)
u = 0 .. 2pi
v = 0 .. pi
hue = 0.55+0.15cos(4u)
size = 0.035
count = 7000
```

Run it (a labs toy) About the shape

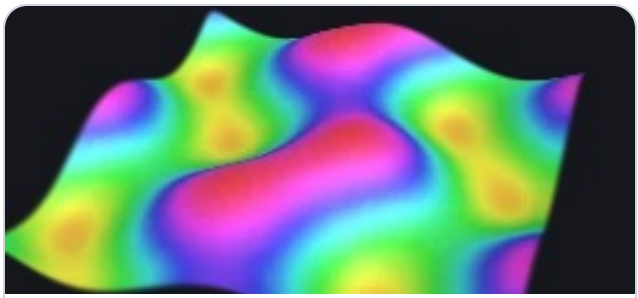


Lemniscate tube

A tube wraps around a sideways figure eight with a subtle vertical wave.

```
x = cos(u)/(1+sin(u)^2)
y = 0.16sin(3u)
z = sin(u)cos(u)/(1+sin(u)^2)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.05
count = 3000
```

Run it (a labs toy) About the shape



Gyroid slice

A height graph of one slice of the gyroid function rises and falls in a repeating pattern of hills and valleys.

```
x = u
y = 0.5(sin(u)cos(v)+0.36sin(v)+0.93cos(u))
z = v
u = -1.5pi .. 1.5pi
v = -1.5pi .. 1.5pi
hue = 0.55+0.3(sin(u)cos(v)+0.36sin(v)+0.93cos(u))
size = 0.11
count = 10000
```

Run it (a labs toy) About the shape



Torus knot family

A (2,3) torus knot winds twice around the ring and three times around its tube.

```
x = (1+0.35cos(3u))cos(2u)
y = 0.35sin(3u)
z = (1+0.35cos(3u))sin(2u)
u = 0 .. 2pi
v = 0 .. 1
hue = u/(2pi)
size = 0.05
count = 3500
```

Run it (a labs toy) About the shape

Things to try. Change a number and see what moves. Swap `cos` and `sin`. Put *t* somewhere new. Make `count` small and `size` big to see the single splats. Try `spread = random` on a surface. Give a sphere a `hue` of `floor(8v/pi)/8`.

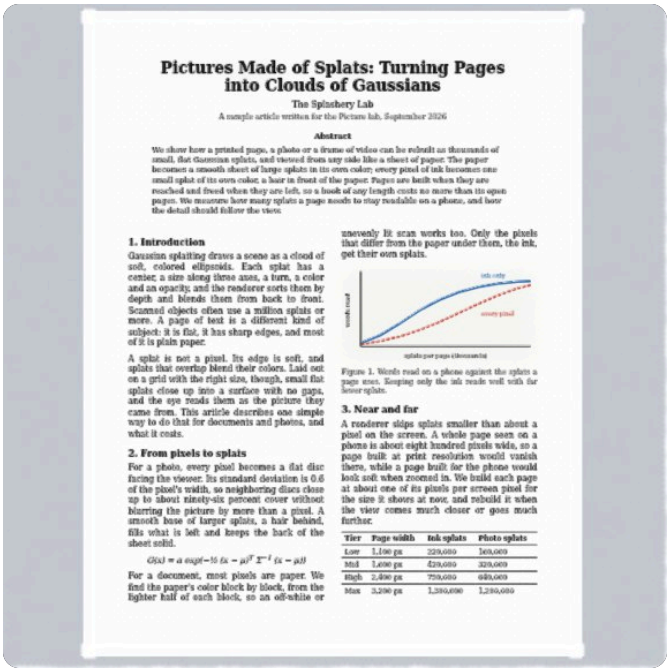
CHAPTER 5

Pictures, screens, and your own input

The toys in Chapter 2 are built from shapes. Some toys show something that is not a shape: a page of a PDF, a photo, a GIF, a video, or something you type or draw. Three additions to a recipe make them: **picture sheets**, an **input panel**, and a few **view settings**. They are still Gaussian splats, all the way down. The toys on the Pictures and pages shelf are being tried out, so they are on the shelf only when Splashery is opened with `?labs=1`.

Picture sheets: a page, a photo, a GIF, a video

A **sheet** is a flat place in a toy where a picture shows: `k.sheet({ id, center, width, height })`. The kit draws nothing there. The site opens the picture before the build, then turns it into splats on the sheet after the build, in a background worker so the page stays smooth. A PDF page becomes a smooth sheet of large soft splats in the paper's color, with a small splat of its own color wherever the page differs from the paper, which is the ink (`method: "ink"`); in a photo, each sampled pixel becomes a flat splat (`"pixels"`); a GIF or a video becomes a screen that is redrawn on every frame (`"screen"`). The site picks the method from the kind of file. It rebuilds a sheet finer as you zoom in, and frees the pages you have left, so a book of 200 pages does not need 200 pages of splats.



Pictures Made of Splats: Turning Pages into Clouds of Gaussians

The Splashery Lab

A sample article written for the Picture lab, September 2026

Abstract

We show how a printed page, a photo or a frame of video can be rebuilt as thousands of small, flat Gaussian splats, and viewed from any side like a sheet of paper. The paper becomes a smooth sheet of large splats in its own color; every pixel of ink becomes one small splat of its own color, a hair in front of the paper. Pages are built when they are reached and freed when they are left, so a book of any length costs no more than its open pages. We measure how many splats a page needs to stay readable on a phone, and how the detail should follow the view.

1. Introduction

Gaussian splatting draws a scene as a cloud of soft, colored ellipsoids. Each splat has a center, a size along three axes, a turn, a color and an opacity, and the renderer sorts them by depth and blends them from back to front. Scanned objects often use a million splats or more. A page of text is a different kind of subject: it is flat, it has sharp edges, and most of it is plain paper.

A splat is not a pixel. Its edge is soft, and splats that overlap blend their colors. Laid out on a grid with the right size, though, small flat splats close up into a surface with no gaps, and the eye reads them as the picture they came from. This article describes one simple way to do that for documents and photos, and what it costs.

2. From pixels to splats

For a photo, every pixel becomes a flat disc facing the viewer. Its standard deviation is 0.6 of the pixel's width, so neighboring discs close up to about ninety-six percent cover without blurring the picture by more than a pixel. A smooth base of larger splats, a hair behind, fills what is left and keeps the back of the sheet solid.

$$G(x) = a\,\exp\!\left(-\tfrac{1}{2}\,(x-\mu)^{T}\,\Sigma^{-1}\,(x-\mu)\right)$$

For a document, most pixels are paper. We find the paper's color block by block, from the lighter half of each block, so an off-white or unevenly lit scan works too. Only the pixels that differ from the paper under them, the ink, get their own splats.

Figure 1. Words read on a phone against the splats a page uses. Keeping only the ink reads well with far fewer splats.

3. Near and far

A renderer skips splats smaller than about a pixel on the screen. A whole page seen on a phone is about eight hundred pixels wide, so a page built at print resolution would vanish there, while a page built for the phone would look soft when zoomed in. We build each page at about one of its pixels per screen pixel for the size it shows at now, and rebuild it when the view comes much closer or goes much further.

Tier	Page width	Ink splats	Photo splats
Low	1,100 px	329,000	369,000
Mid	1,600 px	429,000	329,000
High	2,400 px	729,000	689,000
Max	3,200 px	1,329,000	1,289,000



A PDF page (the Picture lab)

The recipe below: a photo in a frame

Both pictures are made of splats. Photo: "Tulip field" by DennisM2, CC0. The article is our own sample text.

Three parts of a recipe work together:

- `pictures: { sample, accept }` names what shows until you open your own (the sample's address) and which kinds a visitor may open: `"pdf"`, `"image"`, `"gif"`, `"video"`.
- `drive` may choose the page of each sheet with `out.sheets[id] = { page, visible }`. Pages count from 0, and a page that does not exist shows nothing. A sheet that is not visible is not built unless it requests `ahead: 1`. Without it, every sheet shows the current page.
- `info.data.pictures` is the toy's view of the media: `page`, `count`, `kind`, `name`, `playing`, and the functions `next()`, `prev()`, `go(n)`, and `togglePlay()`.

A whole picture toy, in a few lines. Its `input` gives the Toy tab a button that opens a photo, a GIF, or a video; the rest of the toy is a card and four wooden bars:

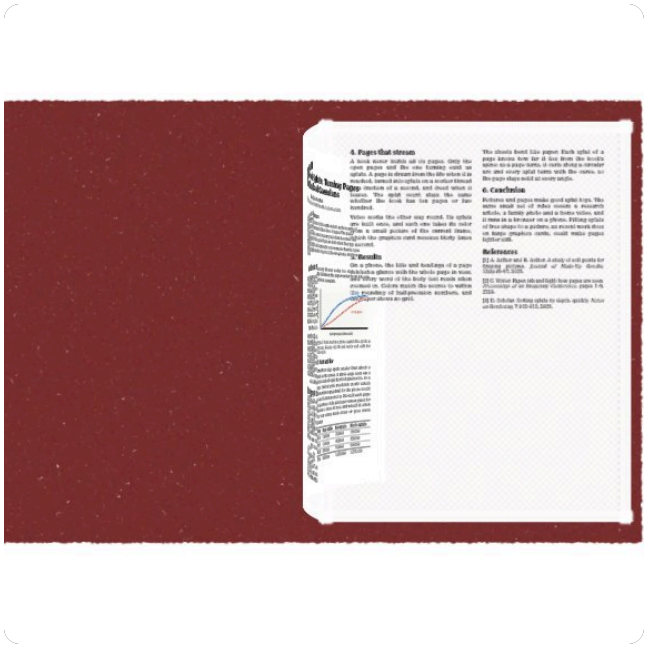
```
import { smoothstep } from "../src/kit.js";

// Two small picture toys for Chapter 5 of the Tinkerer's Manual: a framed photo
// that opens your own picture, and a little book whose page turns like paper.
export const RECIPES = {
  "little-frame": {
    turntable: false,
    tiltLock: true, // a drag only spins it left and right
    pictures: {
```

```
      sample: () => "assets/toys/picture-lab/photo.jpg", // shown until you open your own
      accept: ["image", "gif", "video"],
    },
    input: {
      title: "Your own picture",
      media: { accept: ["image", "gif", "video"] },
      note: "Open a photo, a GIF or a video. Files stay on this device.",
    },
    build(k) {
      // The picture: a sheet, filled with splats after the build.
      k.sheet({ id: "photo", center: [0, 0, 0.03], width: 1.5, height: 1 });
      // A gray card behind it, and four wooden bars around it.
      k.add(k.box(1.6, 1.1, 0.02), { pos: [0, 0, -0.02], color: "#9a9a9a" });
      for (const [w, h, x, y] of [
        [1.8, 0.1, 0, 0.6],
        [1.8, 0.1, 0, -0.6],
        [0.1, 1.1, -0.85, 0],
        [0.1, 1.1, 0.85, 0],
      ])
        k.add(k.box(w, h, 0.1), { pos: [x, y, 0], color: "#8a5a34" });
    },
  },
```

Turning a page

A page that turns like paper is a sheet with `leaf: slot` (a slot from 0 to 9) and a spine to turn about: `k.spine({ at, axis, dir })` gives the spine's place, its direction, and the direction from it along the pages when they lie flat. Each frame, `drive` sets `out.leaves[slot] = { angle, curl }`: the `angle` in radians (0 lying flat, π turned all the way over) and the `curl`, how much the page bends along a circular arc (a negative number lets the free edge lag behind, like paper in air). Every splat of the page turns with the curve at its place, so it stays a solid sheet at every angle, which a page bent with soft regions would not.



The little book below, halfway through a turn. The next page is already there under the turning one.

The little book below uses a pulse control for the turn, the way Chapter 2's gust does. The page under the turning one is built ahead of time (a sheet is normally built only while it is visible, unless it requests `ahead: 1`, so a page that shows up halfway could make the turn wait), and the page number moves on when the turn ends. Keep a turning sheet on the body (part 0), because a part's own turn would replace the leaf's. The far side of the turning page shows the page in mirror image, like tracing paper, until it lands.

```
"little-book": {
  turntable: false,
  tiltLock: true,
  pictures: { sample: () => "assets/toys/picture-lab/article.pdf", accept: ["pdf"] },
  input: { title: "Your own book", media: { accept: ["pdf"] }, note: "Open a PDF." },
  controls: [{ key: "turn", label: "Turn the page", type: "pulse", ease: 1.2 }],
  action: { key: "turn", label: "Turn the page" },
  drive(t, c, out, info) {
    const pics = info.data?.pictures; // { page, count, go(n), next(), prev(), ... }
    // The page changes when the turn ends, and the turning page lands on the left.
    const turning = c.turn > 0;
    if (BOOK.turning && !turning && pics) pics.go(pics.page + 1);
    BOOK.turning = turning;
    const page = pics?.page ?? 0;
    // The pulse falls from 1 to 0: the page swings from 0 to a half turn (pi)
    // about the spine, and its free edge lags behind, like paper.
    const s = turning ? smoothstep(0, 1, 1 - c.turn) : 0;
    out.leaves = [{ angle: Math.PI * s, curl: -1.2 * Math.sin(Math.PI * s) }];
    out.sheets = {
      // Once the turn starts, the left page is the one that is about to land on it.
      left: { page: turning ? page : page - 1 }, // a page out of range shows nothing
      under: { page: page + 1 }, // built ahead, and uncovered as the page lifts
      front: { page }, // the turning page (its far side shows in mirror image)
    };
```

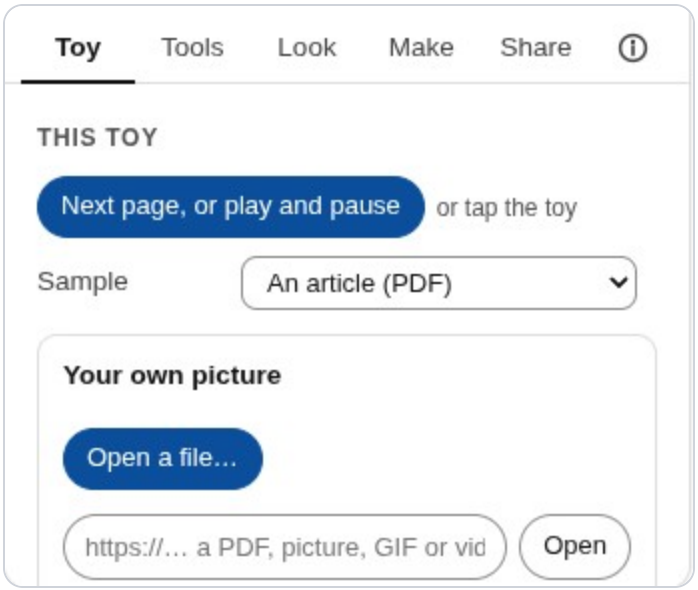
```
      // It is built, but hidden, until the turning page lands on it.
      out.parts.leftPage = { visible: turning ? 0 : 1 };
    },
    build(k) {
      BOOK.turning = false;
      k.spine({ at: [0, 0, 0], axis: [0, 1, 0], dir: [1, 0, 0] });
      const W = 1.4;
      const H = 1.9;
      const leftPage = k.part("leftPage");
      const page = { width: W, height: H };
      k.sheet({ id: "left", ...page, part: leftPage, center: [-W / 2, 0, 0], align: [1, 0] });
      k.sheet({ id: "under", ...page, center: [W / 2, 0, 0], align: [-1, 0] });
      // The turning page: a sheet on leaf slot 0, a hair in front of the others.
      k.sheet({ id: "front", ...page, center: [W / 2, 0, 0.008], align: [-1, 0], leaf: 0 });
      // The cover, behind the pages.
      k.add(k.box(2 * W + 0.1, H + 0.1, 0.03), { pos: [0, 0, -0.03], color: "#7a2f2f" });
    },
  },
};

// Whether the page was turning in the last frame.
const BOOK = { turning: false };
```

Your own input: typing, drawing, opening a file

A recipe can ask for something from the visitor. Its `input` adds a panel to the Toy tab, in one of three forms:

- **Typing**: a text box and a button, and maybe a button that opens a text file (`fileButton` , `accept`). The Splat equation toy of Chapter 3 is one; so is the Word vectors toy, where you type *king − man + woman*.
- **Drawing**: `pad: { cols, rows, max, button }` adds a small grid you paint with a soft brush, and a button that hands the drawing to `read` as the text `"pad:…"` (one number for each cell, row by row). The CNN toy's "3D, draw a digit" view reads the digit you draw.
- **Opening a picture**: `media: { accept }` adds an **Open a file…** button, a field for a web address, the page buttons (or play and pause for a video) and a way back to the sample. Chapter 5's picture toys use it.



The input panel of a picture toy, in the Toy tab. Files are read on your device; nothing is uploaded.

For typing and drawing, `read(text, fileName)` turns what came in into option values, or throws an `Error` whose message the panel shows in place; and `shown()` says what is showing now. Keep what goes into options small: options are saved in the toy's link. Here is a whole toy with a typed input, a row of bars for the numbers you give it:

```
// A small toy with its own typed input, for Chapter 5 of the Tinkerer's Manual:
// type some numbers and the toy stands them up as bars.
const LIMIT = 12;
const COLORS = ["#e8384f", "#ffc233", "#1fb59a", "#4dabf7"];

export const RECIPES = {
  "little-bars": {
    options: [
      {
        key: "numbers",
        label: "Your numbers",
        type: "text",
        default: "3 1 4 1 5 9 2 6",
        hidden: true,
      },
    ],
    input: {
      title: "Your own numbers",
      placeholder: "3 1 4 1 5 9 2 6",
      button: "Show them",
      fileButton: "Open a text file…",
      accept: ".txt,.csv",
      note: `Type up to ${LIMIT} numbers from 0 to 20, separated by spaces or commas, or open a text
file with them.`,
      // Turns the typed text into option values, or throws a message that the panel shows.
      async read(text) {
        const list = String(text).match(/-?\d+(\.\d+)?/g) || [];
        if (list.length === 0) throw new Error("Type some numbers, like 3 1 4 1 5.");
        if (list.length > LIMIT) throw new Error(`That is more than ${LIMIT} numbers.`);
```

```
        if (list.some((n) => n < 0 || n > 20)) throw new Error("Use numbers from 0 to 20.");
        return { numbers: list.join(" ") };
      },
      shown: () => "Your numbers",
    },
    build(k, o) {
      const values = String(o.numbers).split(" ").map(Number);
      values.forEach((v, i) => {
        const h = 0.05 + v / 10;
        k.add(k.box(0.14, h, 0.14), {
          pos: [(i - (values.length - 1) / 2) * 0.2, h / 2, 0],
          color: COLORS[i % COLORS.length],
        });
      });
    },
  },
};
```

The Screen and the Gaussian splatting toy

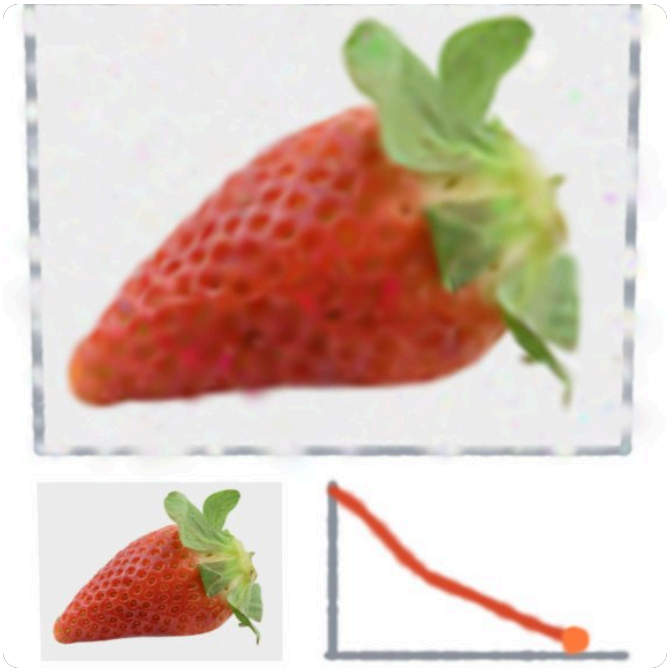
Two more toys put these to work: the Screen on the Pictures and pages shelf, and Gaussian splatting on the AI and computing shelf.

The **Screen** shows a video or a GIF on a screen in one of four styles: an old TV, a flat TV, a cinema with curtains, and a hologram. Its picture is one sheet with `method: "screen"`; the knobs, the curtains, and the glass are ordinary kit parts. A tap switches it on, and then plays and pauses; the speaker button in the top bar turns the sound on. You can open your own video or GIF.

The **Gaussian splatting** toy lets you watch how a photo becomes splats. Its **Training** view fits 2,400 flat (two-dimensional) Gaussians to a photo (a strawberry, or your own), by gradient descent in a background worker: the splats start as random specks and, step by step, they slide, stretch, and change color until together they are the picture. The fit uses front-to-back blending, equivalent to the renderer's back-to-front blending, with the renderer's finite bell, so what you see is the real fit. **One splat** lets you set the sizes, the opacity, and the color of a Gaussian illustrated by a cloud of small splats and turn it (Chapter 1's numbers); **Many splats** builds a small rubber duck from kit shapes and then shrinks every splat to a dot, so you see the splats that make it; and **Sorting** shows the splats of a ball appearing one by one, in the back-to-front order a camera draws them.

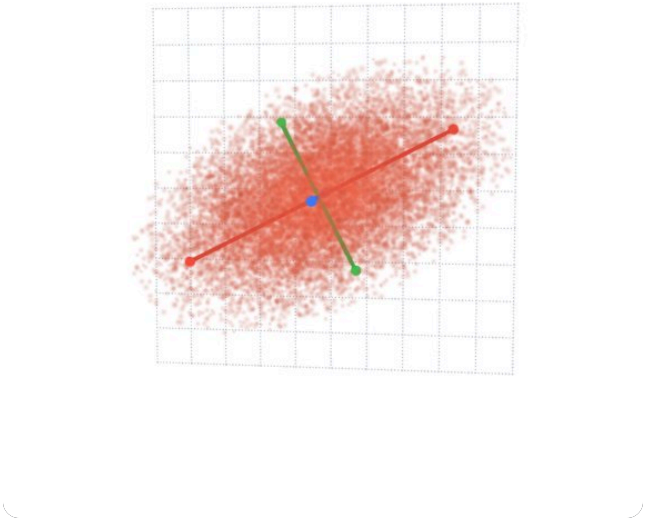


The Screen (old TV)



Gaussian splatting, training

Video frame from "Big Buck Bunny" © Blender Foundation, CC BY 3.0. Strawberry photo by Joselodos, CC0.



The One splat view: a Gaussian illustrated by small splats, with its three sizes drawn as axes (Chapter 1's s_1, s_2, and s_3).

Steering the view: the top bar and the tilt lock

Three round buttons sit in the top bar, beside the sound and help buttons. They control how you look at the toy on show, and none of them changes the toy.



- **Reset view** (the circular arrow) puts the camera back where the toy starts. The **R** key does the same, and so does the button in the Tools tab. A double tap resets the view unless the toy uses it to focus on a page.
- **Tilt lock** (the padlock) makes a drag turn a toy only left and right, so a flat toy such as a page, a photo, or a screen stays facing you. Zooming, panning, and pinching still work. Your choice is remembered for that toy for the rest of your visit. In a recipe, it is the flag `tiltLock: true`, and every toy on the Pictures and pages shelf sets it. A link keeps the camera it was saved with, and the lock then holds that tilt.
- **Turntable** (the record player) switches the slow turn that a toy does when nobody is touching it. It is the same setting as **Turntable when idle** in the Toy tab, and this device remembers it (on a toy that holds still, such as an object or an instrument, your choice lasts only for that toy during your visit). A recipe can opt out with `turntable: false`, as a page does. If your device asks for less motion, the turn stays off and the button is disabled.

Two fingers on a touch screen zoom. They roll the toy only after a clear twist (about 17 degrees) and turn it only when they move together, so a normal pinch never spins what you are looking at. A two-finger drag moves the toy across the screen instead of turning it, as do Shift or Option/Alt and a drag on a computer; ↺ or R puts it back in the middle. A double-tap does too unless the toy uses it for page focus.

CHAPTER 6

Moving things

Chapters 1 to 5 moved splats with programs: a recipe's `drive` sets a part's angle every frame, and the toy plays out. This chapter is about motion that comes from your finger and from physics: pick a toy up, throw it, let it land and settle, open its lid, pull a sword out of a stone. The same rule from the effect-quality notes holds throughout: **parts move as solid pieces**. Nothing is bent. A lid swings on a hinge, a wedge slides out of its orange, and the splats that belong to a part travel with it. This is not behind the labs switch: the ✋ switch is in the Playground for everyone.

Hands-on play: pick up, toss, settle

What it is. With the ✋ switch on, a drag picks up whatever is under your finger. A quick let-go tosses it; it falls, bounces, and comes to rest on the floor under the toy, inside an invisible play area. The ↺ button sends everything home. A press has to move 7 screen pixels before it counts as a drag. On a whole toy, a drag shorter than 24 pixels only *nudges* it (a toy made of pieces, joints, or soft parts is picked up at 7 pixels), and a throw needs at least 24 pixels of travel, so a tap or a light touch never throws anything.

How Splashery does it. Splashery has a small physics engine of its own, with no library (`src/physics/world.js`). It is *extended position-based dynamics* [11, 12]: each step first predicts where the bodies go from their velocities and gravity, then moves them to where they would be if nothing overlapped and every joint held, then works out velocities from how far they moved. The world is built on the first pick-up, so opening a toy costs nothing. It steps 1/60 of a second at a time whatever the frame rate, in 8 smaller substeps for a whole toy and 10 for a toy with pieces, and it falls asleep once everything has been slower than a small speed for half a second.

Most toys play as one rigid body: their pose moves the toy's whole entity, so the splats are sorted and taps land where the toy lies now. A rigid body needs a shape to collide with, and a toy has only splats, so Splashery finds the outside from them. It samples up to 6,000 of the toy's splat centers and, in each of 64 directions, keeps the splat that lies furthest out (ignoring the outermost half of a percent, which are usually stray specks). Those 64 points are the body's collision points. A ball gets a sphere instead, so it rolls. The floor is under the toy; four invisible walls sit just past its widest reach; a ceiling sits above the highest point. Gravity is 26 toy radii per second squared, so a toy falls the same way whatever its size on screen. A throw takes its speed from the last tenth of a second of finger motion, and its base speed is

capped at 4 toy radii per second (the toy's material then scales it, so a baseball or a paper plane can reach 5).

Soft toys squish when they land. How much is a number per toy (61 toys have one, such as the jelly at 1, a bouncy ball at 0.55, a strawberry at 0.15), and it reaches the screen as two uniform values that squash and bulge the toy as a whole. This is a squash of the *pose*, not a bend of the scan: the splats stay where they are inside the toy's own frame.



A rigid body: the yo-yo



Arms that ride the body: the octopus

Both toys can be picked up and thrown in the Playground.

Show the table: the numbers of hands-on play

Number	Where	What it does
7 px	`PRESS_MOVE`	How far a press moves before it is a pick-up.
24 px	`NUDGE`	A drag shorter than this only pushes a whole toy (pieces, joints, and soft parts are picked up at 7 px). A throw needs at least this much travel.
26 radii/s²	`GRAVITY`	Gravity, in toy radii, so every toy falls alike.
0.1 s	`THROW_WINDOW`	The span of finger motion a let-go takes its speed from.

Number	Where	What it does
4 radii/s	`THROW_MAX`	The base cap on a throw, before the material scales it (up to 5 radii/s). Slower than 0.4 (`THROW_MIN`) is a still hold.
0.6 per s	`AIR_DRAG`	Air drag on a thrown toy that has no material of its own.
1/60 s	`STEP`	The world's step, whatever the frame rate.
0.45 s	`HOME_SECS`	How long ↺ takes to glide everything home.
64 directions	`hull()`	A toy's collision points: the farthest of up to 6,000 sampled splats per direction.

Why toy radii, and why a speed limit?

OUR CHOICE Nothing in the format forces these.

Splashery's toys are an inch tall or a planet wide, and the page has no real size. So the world measures everything in the toy's own radius: gravity, the throw's speed, the play area. A matchbox and a mountain then fall and land the same way, and a throw cannot send a toy off the screen. The cost is that the motion is not physically scaled: a real baseball is thrown a hundred times faster than 4 of its radii a second. Where that matters (the curve of a spin, the glide of a disc), the numbers are set to show the real effect in the real order, as the section on materials below says.

Where it lives: the constants at the top of `src/physics/hands-on.js` .

Pieces and joints: parts that move the way the real thing's do

What it is. A rigid body is one lump. A treasure chest has a lid, and a lid is not a lump: it swings on its hinge and stops when it is open. A recipe's `hands` block lets a toy say that. `hands.pieces` names pieces of the toy that are bodies of their own, which stay put until you pick one up or knock it. `hands.joints` says how a part may move.

How Splashery does it. There are five kinds of joint and one more setting (`src/physics/joints.js`). Hinges, sliders, and dials are *driven parts*: one number describes each, an angle or a distance. The finger, gravity, a spring, and the stops move that number, and the part is then posed exactly from it, so a part never wobbles off its axis or jitters. Sockets and broken-off pieces are ordinary bodies of the physics world, solid like the rest.

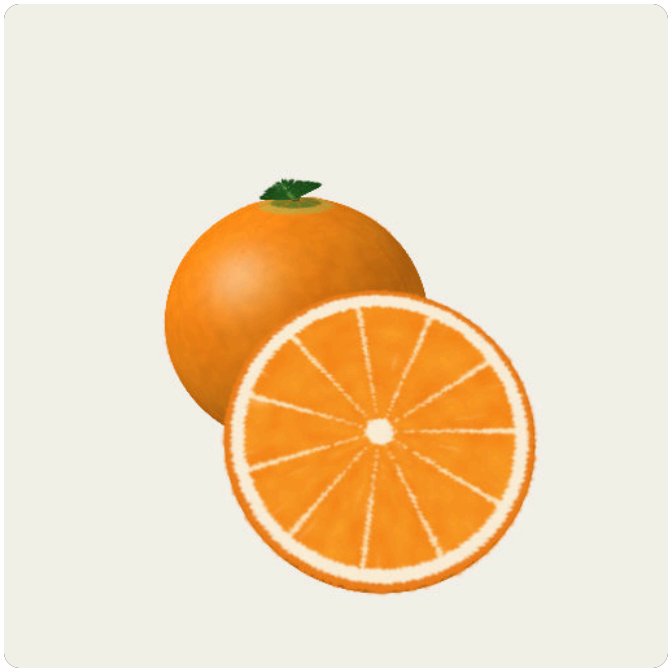
- **Hinge**: a part swings about an axis between two angles, with an optional spring toward a rest angle, and falls by its own weight. It bounces off its stop a little (0.2 of the speed, by default). The treasure chest's lid is one.
- **Slider**: a part moves along a line between two stops. It can be stuck fast until the finger has pulled a set distance, and then it wiggles loose, as the sword in the stone does.
- **Dial**: you turn a part about its axis, round and round or between two angles, with optional clicks (detents) and coasting after a flick. A dial can hand each turn to the recipe, which is how the music box's crank plays its notes.
- **Socket**: a loose piece clicks back into its place when you bring it near (within 0.3 toy radii, by default), gliding the last bit in 0.22 seconds. The wedges of the orange are sockets.
- **Break**: a piece holds fast until it is pulled hard, then snaps off. ↻ mends it. The candy cane breaks this way.
- **Upright**: not a joint but a setting, `hands.upright`: a whole toy rights itself after it has been tipped.



A hinge: the chest's lid



A slider: the sword



Sockets: the wedges

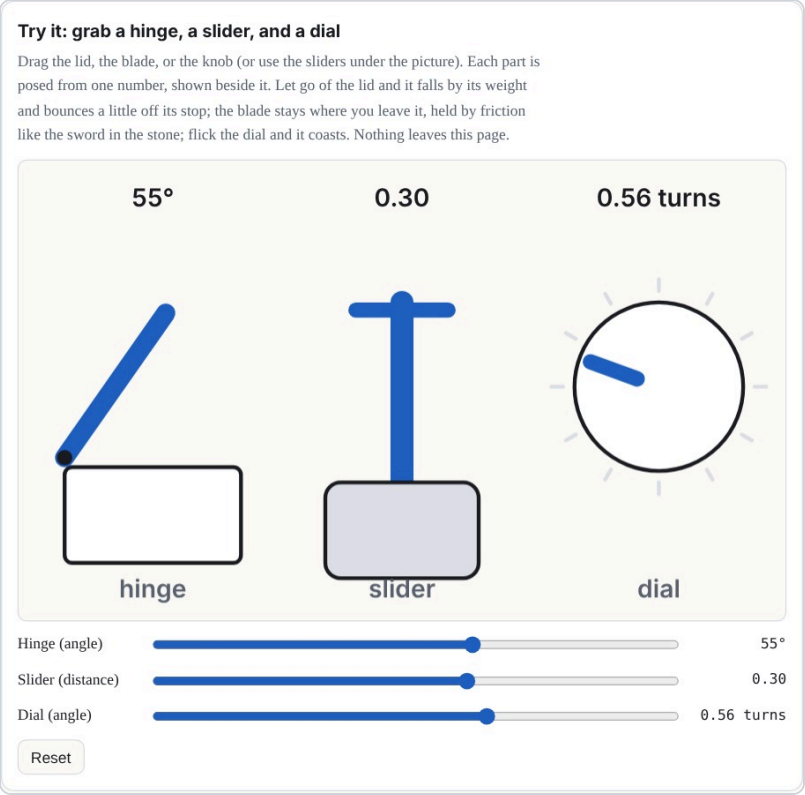


A break: the cane's top

Each part is a solid group of splats that moves whole. The music box's dial is on the shelf too.

Why this way. Splashery's rule for effects is that motion must be real motion and not a warped picture: if a scan cannot move a part cleanly, the part is cut out with hard edges, swapped for a kit-built one, or the effect is dropped. A joint that moves one number, and a part that is moved whole by that number, is exactly that rule in

code. The same machinery serves a scan: a scan's rig (`src/rigs.js`) names the parts of a captured toy, and `src/physics/joints.js` turns them into pieces.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/moving.html#demo-joints

Ropes, cloth, and soft stretch

What it is. Some things bend: a string, an octopus's arms, a hood, a kite's tail. Splashery's rule against bending scans does not forbid them, because these toys are built from kit shapes and their bending is built in.

How Splashery does it. Ropes and cloth are lists of points (*nodes*) held by distance links, solved with the same position-based method as the rigid bodies, in a solver of its own (`src/physics/soft.js`). A rope can be slack (it folds and never pushes), has a little stiffness against bending, and can be pinned, can ride a piece (an octopus's arm rides its body), or can follow a finger. A cloth is a grid of nodes, pinned at its edge, that hangs, drapes, and flutters in a wind. A link can snap when it is stretched too far, as a cheese string does. Soft stretch, the third piece, is how the gummy bear's splats follow your finger and then wobble back.

What moves on screen follows the rule too. A thin, bendy thing (a string, a tentacle) is built from splats that each follow the two nodes they lie between, so a rope bends smoothly and every splat still keeps its shape. A sheet has each splat follow the four

corners of its cell. A solid thing on a rope (a bead, a chime, the yo-yo itself) is a rigid piece that rides a node and turns with the link, so it stays solid at every angle.

Try it. Pull the string of the yo-yo and let go, or hold the octopus by one arm.

Materials: a thrown thing moves like the real thing

What it is. A basketball, a hockey puck, a paper plane, and a marble do not move alike. A recipe's `hands.material` names a preset or gives its own numbers, so a thrown toy bounces, rolls, spins, and flies the way its real counterpart does.

How Splashery does it. There are 27 presets (`src/physics/materials.js`), from the basketball and the soccer ball to the shuttlecock, the flying disc, and the paper plane. A preset holds the real regulation size and mass of the sport, the bounce its rule book asks for (a basketball dropped from 1.8 m should come back to 1.2 to 1.4 m, which is a bounce of 0.82 to 0.88), and the textbook drag of a sphere (0.47). The drag is the real thing, expressed without units: how many toy radii a toy falls before it reaches its top speed, $F_r = 2m / (\rho\, c_d\, \pi\, r^3)$ with air at 1.2 kg per cubic meter. Because that number has no units, it holds whatever the toy's size on screen. A throw spins the toy by where you held it (a low grab gives backspin, a side grab sidespin), and the spin curves the flight (the Magnus effect). Lift, for a disc, a cap, or a paper plane, and the curve of a spin are set to show in a short flight rather than scaled from real life, in the real direction and the real order: a disc glides more than a cap, and a baseball curves more than a soccer ball.

Try it. Throw the basketball, the baseball, and the paper plane and watch how each leaves your hand.

Rigid parts and the motion driver

What it is. Everything above, and everything a recipe's `drive` does, ends in the same place: a few numbers per part for the graphics chip.

How Splashery does it. The `MotionDriver` in `src/motion.js` runs once per frame, on the CPU. It eases the controls, runs the toy's own behaviors clock, calls the recipe's `drive` , and gathers what comes out into uniforms. For parts, `packParts` writes three four-number values for each of up to 16 parts (a rotation, a pivot with a scale, and an offset with a visibility), 48 in all. Every splat carries the number of its part; the shader looks up the part and moves the splat: p = pivot + R (p − pivot) × (1 + scale) + offset. This is the same hinge as Chapter 2's, and a part that a joint moves goes through the very same lines. Hands-on does not have a second way

to move a splat: for a toy with pieces, the poses its world computes are merged into the recipe's `drive.parts`, so a piece moved by a finger and a part turned by a program arrive at the shader identically.

Why this way. The GPU moves every splat, and the CPU only decides a few dozen numbers: a physics world with a few dozen bodies costs almost nothing, and a toy with 100,000 splats moves as cheaply as one with a hundred. It also keeps the splats solid: a part's splats are all turned by the same rotation, so a part cannot bend. The limit is that a splat can belong to only one part, which is why a toy has at most 15 parts beside the body (Chapter 2's fold on the number of parts says why).

Where to try it

- Treasure chest, Sword in the stone, Music box, Orange, and Candy cane: a hinge, a slider, a dial, sockets, and a break. Turn on the ✋ switch first.
- Yo-yo, Octopus, and Gummy bear: ropes, arms that ride a body, and soft stretch.
- Basketball, Baseball, and Paper plane: materials.

None of these is behind the labs switch. Chapter 10 says how a recipe asks for hands-on play.

CHAPTER 7

Real data as splats

A splat is a number-carrying blob: a place, three sizes, a turn, a color, an opacity. Data is also numbers with places: the atoms of a protein, the quakes of last week, the stars over your head, the density of a scanned walnut. This chapter is about turning such data into splats, and doing it honestly: every toy here says where its data came from and how old it is. Nearly all of these toys are behind the labs switch (add `?labs=1` to the site's address); the paragraphs say which.

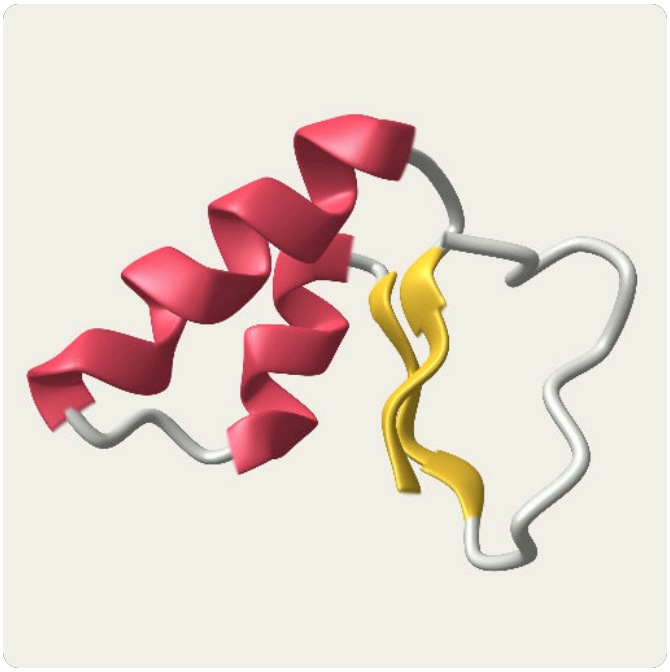
Molecules

What it is. A molecule is a list of atoms with places, in ångströms. The **Molecule viewer** opens a structure file and draws it as splats. It reads PDB, mmCIF, SDF/MOL, and XYZ files (`src/molview/parse.js`), takes the first model, and stops with a message above 250,000 atoms. You can open a file from your device, or type a four-character Protein Data Bank code and have the viewer fetch that entry. It draws balls and sticks, space-filling atoms, a cartoon of the chains as ribbons, or the molecular surface, and colors by element, chain, residue, B-factor (how much an atom blurs in the crystal), position along a chain, or secondary structure. Tap two atoms for the distance between them, a third for the angle. It is behind the labs switch.

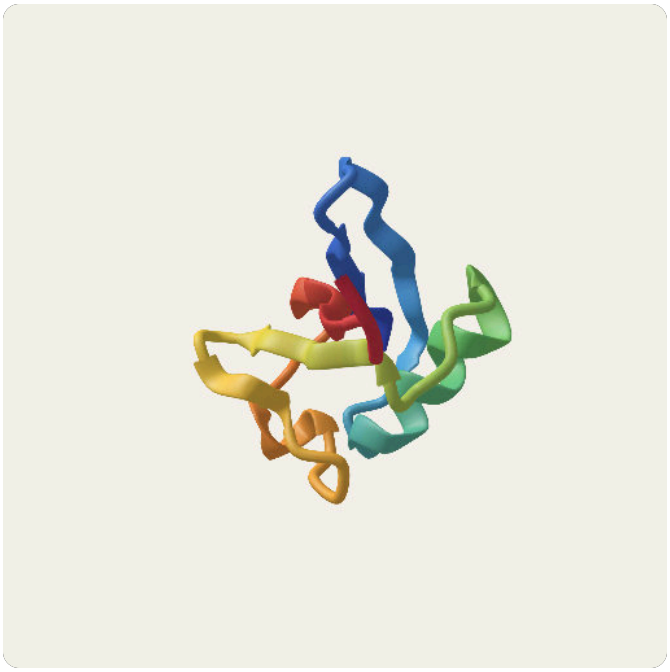
How Splashery does it. Everything is an explicit splat (`src/molview/draw.js`): an atom is a sphere of splats (in space-filling style, one of its van der Waals radius), a bond is a stick of splats (a double bond is two thinner sticks, a triple bond three), the cartoon is a ribbon of splats along a path through the backbone, and the surface is a blobby surface made from a sum of bumps, one per atom. The device's splat budget decides how many splats there are. A small molecule gets plenty per atom. A big structure gets fewer, and below about six splats an atom becomes one round Gaussian, so even a big structure opens on a phone: that is the level of detail (Chapter 2). The files are read in a worker, off the page's main thread, and the structure is turned so its widest spread lies across the screen. A turn keeps every distance and angle.

The PDB fetch. Typing a code asks `files.rcsb.org` for that one file (public, no key, no account), only when you press the button. Nothing is sent but the code in the address, nothing is stored, and the file is read on your device. The viewer refuses a file over 60 MB. Five samples ship with the site, so the toy opens offline: crambin, the green fluorescent protein, lysozyme, a stretch of B-DNA, and caffeine; each is a

snapshot of the entry, dated October 5, 2026, and the data are CC0 from the Protein Data Bank [14].



The Molecule viewer's cartoon



The Protein toy, on the Atoms shelf

Both are drawn with splats. The Protein toy is one of the always-on Atoms toys, and the viewer is the labs version that opens your own file.

Try it. Molecule viewer (labs), Protein, Molecule.

Earthquakes

What it is. A globe with a dot for every earthquake in a feed from the U.S. Geological Survey, drawn on a relief of the Earth. A bigger quake is a bigger dot. The color says how deep the quake was: shallow ones orange-red, middle ones yellow and green, and the deepest blue, as the USGS's own maps do. The globe is the Earth in true color with its relief. A tap plays the quakes in time order, each flashing as the timeline passes it.

How Splashery does it. The toy offers three views: the past week at magnitude 2.5 and up, the past month at 4.5 and up, and a year at 5 and up. The first two read the USGS's GeoJSON summary feeds [15] when the toy opens or when you tap its plaque to refresh. Each quake is a row of five numbers (time, longitude, latitude, depth in kilometers, magnitude). A quake with a missing magnitude or depth is left out rather than shown as zero. A dot's radius is $0.009 + 0.006 \times (M - 2)^{1.4}$ times the globe's radius (for a magnitude of 2 or more), drawn as a flat disc of splats with a

crisp edge, and a bigger dot gets more splats, in proportion to its area. The plaque beside the globe says what is showing: the feed, the number of quakes, and when it was made. If the feed cannot be reached within 9 seconds, or if the browser is being driven by a test tool, the toy shows the snapshot that ships with the site (197 quakes in the week view as of October 5, 2026, 14:39 UTC), and the plaque says "snapshot". The year view is always a snapshot, October 2025 to September 2026.

Why this way. Splashery has no server, so a live feed has to be read by your browser from a public, keyless address, and it has to degrade: an offline reader still gets a globe. The rules for every such toy are in the box at the end of this chapter.



The earthquakes globe on its snapshot, with the plaque that names the source and the time.

Try it. Earthquakes (labs).

The night sky

What it is. The stars and planets above a place, for any date and time and any place. You stand at the center of the dome and look out; the Sun, the Moon, and the seven planets are placed from the date (the planets' tables are made for 1800 to 2050, and the toy says so when you go outside them), the Moon is lit on the side that faces the Sun (so its phase is the real geometry), and daylight washes the stars out faintest first.

How Splashery does it. The star catalog is a file of 5,071 stars: every star brighter than magnitude 6 in the HYG database [16], plus the fainter ones a constellation line needs. Each displayed star gets a main splat, sized by brightness and colored by temperature, and the brightest also get a larger halo splat. The stars sit at their J2000 positions as one part, and the whole sky turns as one rigid piece: first precession to the date, then the sky's turn for the place's sidereal time (Meeus [17]). The planets use the approximate orbital elements that NASA's Jet Propulsion Laboratory publishes for 1800 to 2050 [18], solved with Kepler's equation, and the Moon uses the Astronomical Almanac's low-precision series, good to about 0.3 degrees in longitude. This is the same hinge as every part in Chapter 2. A star field turning about a point is a part turning about its pivot.

Your place. You pick a city, or tap **Use my location**, which asks your browser for it. That is the only time the toy asks. The place lives in the page's memory: it is not saved, not put in a link, and not sent anywhere. Both the stars and the constellation lines come from sources under CC BY-SA 4.0, and the toy's credit says so beside it.



The night sky toy at dusk, with constellation lines.

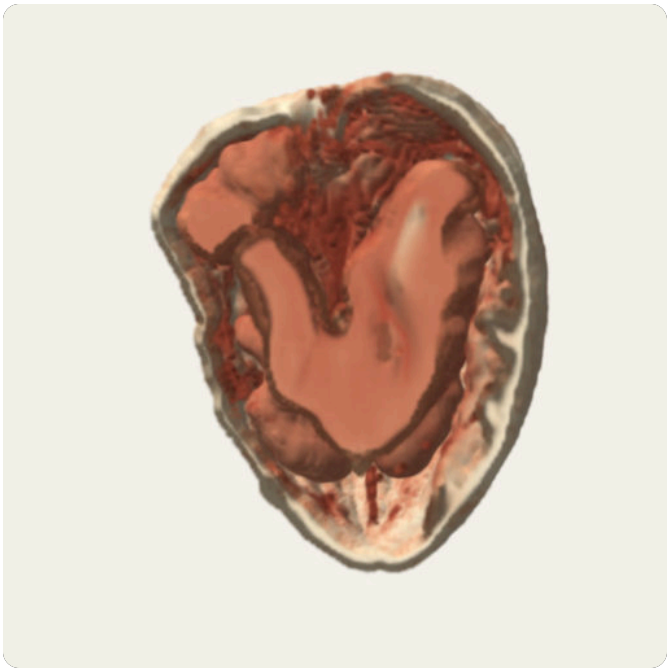
Try it. Night sky (labs).

Volumes and point clouds

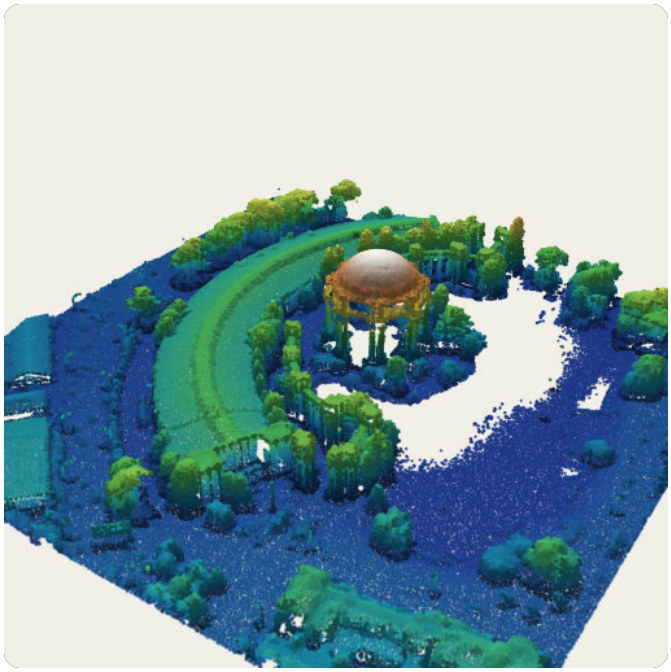
What it is. A CT scan or an MRI is not a surface but a block of numbers, one per little cube (a *voxel*), measuring how dense the material is there. A lidar scan is a cloud of points with places and sometimes colors. Both can be splats. The **Volume viewer** opens a medical or microscope scan and shows it as a block of splats you can cut through. The **Point clouds** toy opens a lidar scan or any list of points. Both are on the Studio shelf, behind the labs switch, and nothing is uploaded.

How Splashery does it: volumes. The viewer reads DICOM (a series of slices, a folder, a zip, or one multi-frame file; uncompressed or RLE lossless), NIfTI (`.nii` and `.nii.gz`), TIFF stacks, and raw files described by a small form (`src/volume/read.js`). Slices are sorted by where the scanner put them, and the spacing between slices comes from those places, so a scan that is thick in one direction keeps its true shape. A volume is at most 8 million voxels in memory; a bigger one is averaged down as it is read, so the density grid it keeps is never more than about 32 MB (16 MB on a phone-sized device). Reading the file and drawing the splats use more memory besides. The splats sit on a lattice spaced to fit the device's budget. A density **window** (a lower and an upper level) decides which values show; the presets are bone, soft tissue, and full range, and for a CT in Hounsfield units bone is 300 to 1500 and soft tissue is −160 to 240. A color map turns the value into a color and an opacity. A cutting plane on each axis, which a drag moves, hides the splats on one side. A **maximum-intensity** view flattens the volume to the brightest voxel along one axis. Splashery's engine has a splat kind for this (`volume`): each splat carries its density, the shader tests it against the window, and the cut is tested at the splat's place at rest, so the cut stays with the specimen as it turns.

How Splashery does it: point clouds. The toy opens LAS, LAZ, PLY, XYZ, and PTS files (LAZ through the vendored laz-perf decoder, loaded only for a LAZ file). The points are drawn as splats within the device's budget, colored by height, by intensity, by classification (ground, vegetation, buildings, and the other codes a lidar file uses), or by the file's own colors. You can measure between two taps, crop, thin the cloud, and save it as PLY, LAS, or XYZ. The three samples are lidar from the U.S. Geological Survey's 3D Elevation Program, public domain: the Palace of Fine Arts and Fort Point in San Francisco, and Meteor Crater in Arizona, each thinned to 450,000 points.



A CT scan of a walnut



Lidar, colored by height

Walnut CT: CWI, CC BY 4.0. Lidar: USGS 3DEP, public domain.

Why this way. A splat can stand for a voxel or a point without any new idea: it is a place, a size, and a color. What the toy has to add is a way to look *inside*, and that is the window and the cutting plane. Rendering a volume with splats gives up the smooth shading of a proper volume renderer, in return for the same engine, the same controls, and a saved link that opens it in the same viewer as everything else.

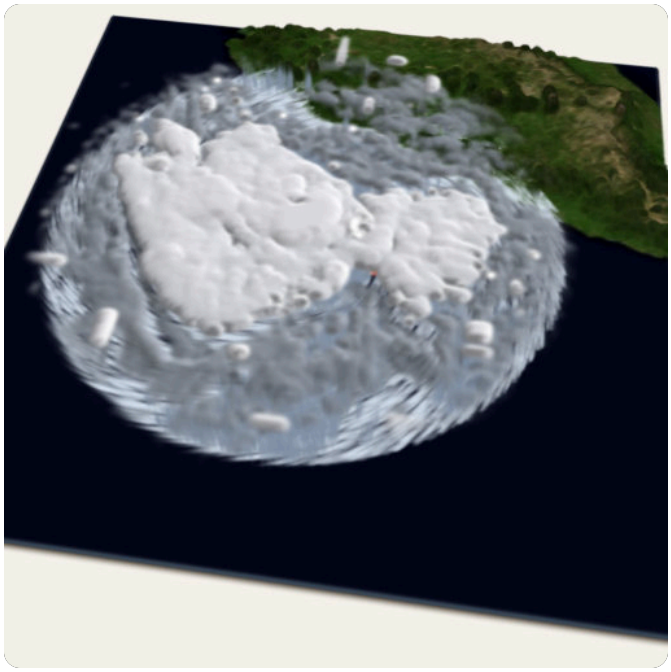
Try it. Volume viewer, Walnut CT scan, Point clouds (all labs).

Weather, maps, and the other data toys

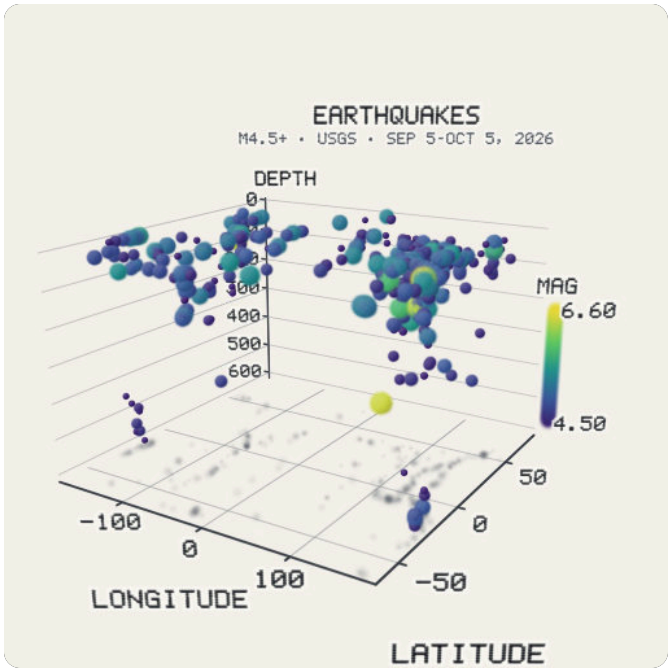
Most of the data toys ship a **dated snapshot** that a tool on a build machine fetched once, so the page itself never calls a data service. The toys are:

- Hurricane (labs): Hurricane Polo, September 21 and 22, 2026. GOES infrared cloud tops over a satellite map of the Earth, with the eyewall and bands turning. A tap plays the day it grew from 70 to 155 knots, following the storm's best track.
- Tides at Bar Harbor (labs): a coastal height map and imagery of the harbor in Maine, with NOAA's tide predictions for a spring tide; a tap plays the day and the sandbar to Bar Island floods and dries.
- Grand Canyon, Mount St. Helens, and The sea floor (labs): heights from the USGS 3D Elevation Program and NOAA's ETOPO1 relief, as terrain made of splats.

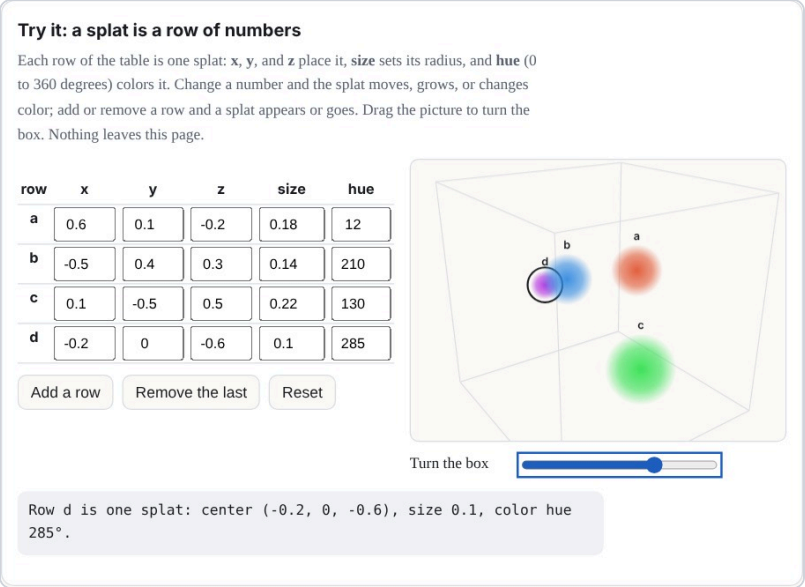
- Stork migration (labs): 30 white storks tagged in Germany, flying their real GPS tracks to Africa in fall 2013 (Movebank, CC0). The track file ships with the site; the tool that downloaded it is in `tools/`.
- Climate records (labs): NOAA's Mauna Loa carbon dioxide record as a rising spiral, one turn a year, and NASA's temperature anomalies as a field of monthly bars or a wall of yearly stripes. Dated snapshots, never fetched live.
- Data in 3D (labs): open a CSV or TSV from your device, pick the columns for X, Y, Z, color, and size, and see a scatter, bars, or a surface with labeled axes that turn to face you. Three small sample tables ship with it. It saves a picture (PNG) or a turning video (WebM) to a file you choose.



Hurricane Polo



Data in 3D: the quakes as a scatter



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/data.html#demo-table

The rules for live data

A toy that reads from outside the page follows a short list of rules:

- Only a public endpoint that needs no key or account, and only data, never code: the earthquake feed, an entry from the Protein Data Bank. (Other open feeds, such as animal tracking, may be read the same way when their terms allow; today the stork tracks ship as a file.)
- It happens when you open the toy or tap to refresh, never before.
- The source and the time of the data show beside it.
- Nothing is stored and nothing is sent, except the request for the data itself.
- A dated snapshot ships with the site, for when the feed cannot be reached.
- The night sky may ask for your location, only when you tap for it, and never stores or sends it.

A **Wikipedia book**, which would fetch an article when the reader asks for it (CC BY-SA text, never stored or shipped), is planned. The rules allow it, but it is not built yet, so it is not in this manual's list of toys.

Where to try it

- Protein and Molecule (always on); Molecule viewer (labs).
- Earthquakes, Night sky, Volume viewer, Point clouds, Hurricane, and Data in 3D (all labs).

CHAPTER 8

Photos and video into 3D

Chapters 6 and 7 start from things that are already numbers: a toy's recipe, a table, a catalog of stars. A photograph is not. It is a flat grid of colors with no depth, and a video is that grid thirty times a second. This chapter is about the ways Splashery gets 3D out of them, from the cheapest guess to a full reconstruction, and about what it can and cannot promise. The tools here are on the Studio shelf and behind the labs switch; the photoreal captures are on their own shelf, partly open.

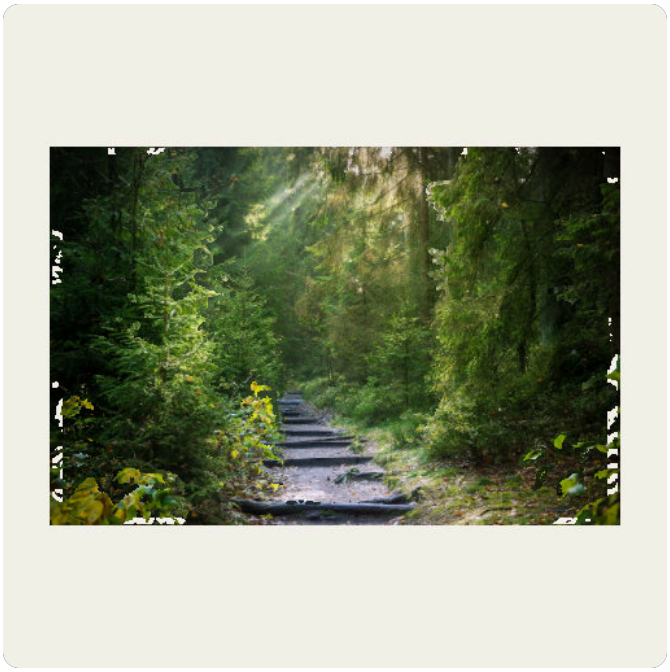
Photo to 3D

What it is. Open a photo and get a toy you can turn: near things move across far things as you turn it. It is a depth effect, a guess, not a scan: from one picture there is no way to know what is behind a tree.

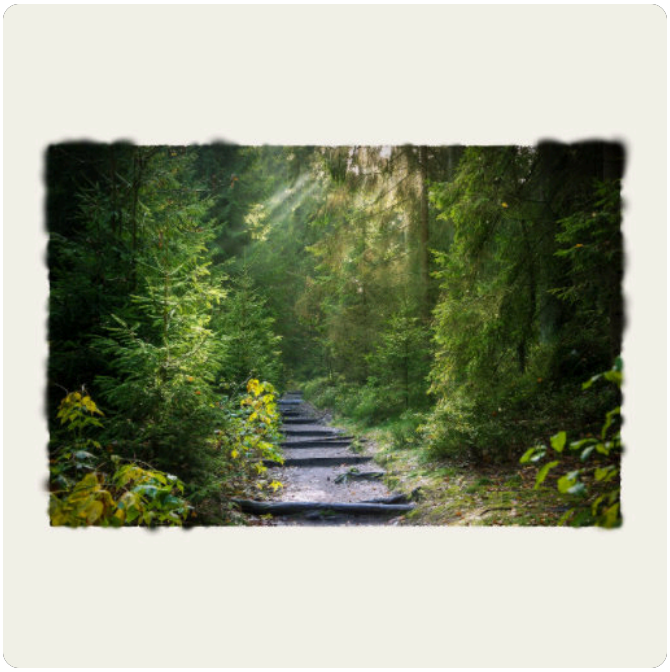
How Splashery does it. A depth model works out, for every part of the picture, how far away it is. The model is Depth Anything V2 Small [13] in its 8-bit quantized form, a file of 27 MB that runs in your browser through ONNX Runtime Web on one thread. (A page on GitHub Pages cannot share memory between threads, so the runtime has just one.) Both files are Splashery's own, served from the site; nothing is fetched from anywhere else and your photo is not uploaded. They load only when you open a photo of your own. The sample pictures on the toy come with their depth maps already worked out, so they need no model at all. Some samples are AI-made pictures, and each one says so beside it.

The picture goes to the model scaled so its long side is about 518 pixels, each side a multiple of 14 (the model reads the picture in 14-pixel tiles). The answer is a smooth map of *relative* depth: it says which parts are nearer than which, not how many meters. The photo is then rebuilt as splats. Each splat sits at its place in the picture and at its depth, colored from the photo, and sized to its neighbors on its own surface so the surface closes. A grid that adapts to the picture spends the splats where there is detail: a plain block of a wall is one big splat, and a block of letters splits into four, and again, down to single pixels, until the budget is spent. The budget is the tier's default count times 1.5, capped at the tier's maximum: 90,000 splats on a low device, 210,000 on a mid one, 300,000 on a high one, and 400,000 on the highest. Where the depth jumps between two neighbors by more than 0.015 of its range, the surface is cut, so a near object stands as its own layer and does not smear into the wall behind it. The picture is split into four depth layers, and a tap lifts them apart one at a time.

Why this way. The depth model needs only one photo and runs on the device, which fits a site with no server. The price is the model's guess. Splashery does not try to paint what the camera did not see, so a turn far enough shows the cuts.



The Sharp view (the default)



The splat view

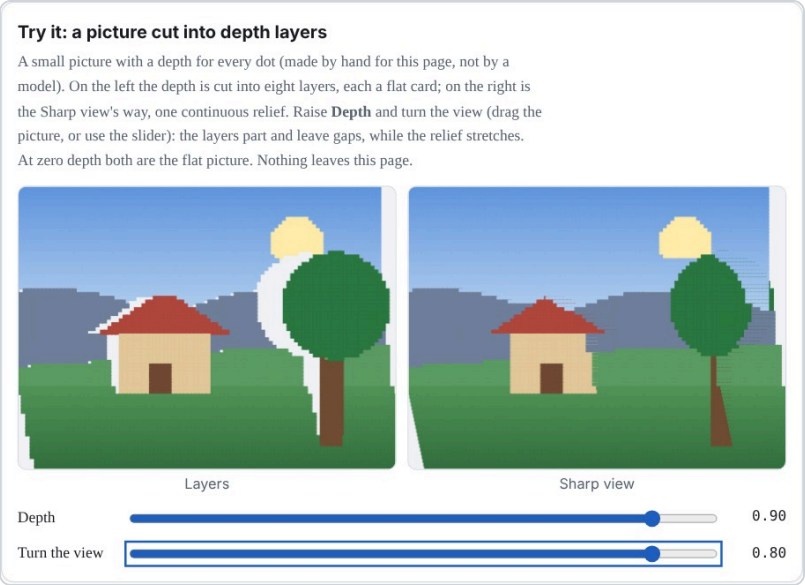
The same photo, two views. Sample photo: "Forest path" by Seaq68 (from Pixabay, marked CC0 on Wikimedia Commons).

The Sharp view

What it is. Splats are soft by nature, and a photo of a page of small print loses its letters when it is drawn as 200,000 soft blobs. The **Sharp** view is a second view of the same toy that keeps the photo at its full resolution.

How Splashery does it. The photo is laid as a texture on a *relief*: a surface lifted by the same depth the splats use. It is a textured surface, not splats. The depth slider, the tap, the layers, the sway, and, for a clip, play, pause, and scrubbing all move the relief as they would move the splats. A switch in the toy's panel chooses Sharp or Splats, and the choice is saved with the scene, so a shared link opens in the view it was saved in. A scene saved before the switch existed opens in Sharp. Where splats are needed, they show instead and the panel says so: while Poke, Paint, Magnet, or Clay is the active tool, in hands-on play, and while a Look effect is on. Saving the toy as a splat file is unaffected, since it reads the splats.

Why this way. Splashery is primarily splat-based but not entirely splat-based, and this is the clearest place it is not. The Sharp view is there because splats are the wrong tool for reading small text, and because the toy is still the same toy: the same depth, the same controls, the same saved link.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/photo3d.html#demo-depth

Moving photo to 3D

What it is. The same idea for a GIF or a short video: it plays back in 3D, each frame lifted by its own depth. The sample is six seconds of "Big Buck Bunny" (the bunny and the butterfly; © Blender Foundation, CC BY 3.0) with its depth worked out ahead of time, so it needs no model.

How Splashery does it. For a GIF (its first 8 seconds) or a video of 8 seconds or less, it takes the frames at up to 12, 15, or 24 frames a second depending on the device tier, each frame sized to 60,000, 140,000, or 200,000 pixels (or 640 by 360 on the highest tier). Running the depth model on every frame would take too long, so it works out the depth on at most 24, 32, or 48 frames spread evenly through the clip, at 196 or 294 pixels on the long side, and the frames between blend their neighbors' depths. The colors carry the motion. A video plays with its own sound and its frames follow that sound's clock. The depth is smooth, since the model guesses shapes and not edges, so it is sampled smoothly up to the frame's size and its edges are cut clean. A video longer than 8 seconds plays whole instead: its depth is worked out a few times a second, in order, while it plays, with the frames between blending the stored depth pictures, and its colors come from the playing video. It has the Sharp view too.

Video to 3D

What it is. Photo to 3D guesses depth from one picture. This toy does something different and harder: it takes *many* views of a still scene (the frames of a video that walks or flies around something), works out where the camera was for each, and trains a real 3D Gaussian splat scene that looks right from each of those places. It is the method of Chapter 1's first reference [1], running in your browser. A scene that stands still works. A moving car or person blurs or vanishes.

How Splashery does it. It uses Splat.js [21], an MIT-licensed library vendored in `vendor/splatjs/` and loaded only when you open a video in this toy. The toy first picks the sharpest frame in each window of the stretch you choose. Splat.js then works out the camera's path (it finds features, matches them between frames, registers each camera, and adjusts them all together), seeds a splat cloud, and trains it on the graphics chip through WebGPU: the splats' places, sizes, colors, and opacities are nudged by gradient descent until the scene renders like each frame. The result is shown as the toy's splats, and you can turn it, zoom, save it as a PLY, or tap **Replay flight** to fly the video's own camera path with its sound.

How much it does depends on the device. A mid-tier device uses up to 32 frames at 640 pixels, trains 3,000 steps, and keeps up to 120,000 splats; a high-tier one uses 60 frames, 7,000 steps, and 250,000 splats; the lowest tier, a phone, uses 16 frames, 1,200 steps, and 30,000 splats, with at most 8 minutes of training (it keeps what it has by then; preparing the frames and cameras takes extra time). It needs WebGPU: a browser without it says so rather than starting. The samples (a statue flown around, a street walked) are results made the same way from two CC BY 3.0 videos; the full videos are not shipped, only the trained splats, the camera path, and a short cropped clip of each (for Show the original).

Why this way. This is the one place Splashery trains a splat scene in your browser. It is honest about the cost: minutes of work on your graphics chip, a result only as good as the video, and no server.



A moving photo



A trained video: the statue sample

Video frame from "Big Buck Bunny" © Blender Foundation, CC BY 3.0. The statue scene is trained from a CC BY 3.0 drone video (credit beside it in the toy).

The photoreal captures

What it is. The Photoreal shelf holds real objects (a strawberry, a honeybee, a sunflower, a bonsai tree) that other people captured with many photographs and trained into 3D Gaussian splat files, which Splashery converts and shows. Of its 72 toys, 64 are captured scans and 8 are kit-built; 32 are open to everyone and 40 are behind the labs switch until they are ready.

How Splashery does it. Splashery does not train these. Each capture arrives as a file under a license it can use, and a build tool converts it: it decimates the splats, centers and scales the toy, and writes a `.sog` file (a compressed splat format), usually as a pair, a full file and a lite one for weaker devices. The viewer opens `.ply`, `.sog`, `.splat`, and `.spz`, so it can also open your own captures. Files can carry spherical harmonics, the view-dependent color the kit's own splats do not use, and some of the newer captures keep one band of them on the full file. Big files warn and then thin: a low device warns at 60 MB or 400,000 splats and thins to 300,000, and the highest tier warns at 200 MB or 2,000,000 splats and thins to 1,500,000.

Licenses. The 64 captured toys carry their credits beside them: 16 are CC0, 44 CC BY 4.0, one CC BY 2.0, and three are NonCommercial (CC BY-NC or CC BY-NC-

SA), which are flagged in the asset list so they can be taken out if the site ever earns money. A capture under a NoDerivatives license, or with no license, is never used.



A strawberry



A honeybee

Photoreal captures. Strawberry: Dany Bittel, CC BY 4.0. Honeybee: YUMA Co., Ltd., CC BY 4.0. Credits and licenses show beside each toy in the Playground.

Where to try it

- Photo to 3D, Moving photo to 3D, and Video to 3D (all labs; the last needs WebGPU).
- Strawberry, Honeybee, and Grape (open to everyone). The grape has a peel that shows the pale flesh.

CHAPTER 9

Codes and pages

A code is a picture that stands for bytes. A page is a picture that stands for words. This chapter is about toys that are, in the end, flat: QR codes and other barcodes, the pages of a PDF, a photo, a GIF, and a link. They are made of splats too, and they show a point that Chapters 1 to 5 made in passing: **a flat thing made of splats is only useful if every splat stays where the picture needs it**. A QR code that scans only some of the time is not a QR code. So the toys here check their own work.

QR codes

What it is. A QR code is a square grid of dark and light cells (*modules*) that holds a few hundred bytes of text, plus extra data that lets a reader rebuild the text if some cells are damaged. The three big squares in the corners (*finder patterns*) let a camera find and orient it, and a blank border (the *quiet zone*) four modules wide keeps the picture from running into whatever is around it.

How Splashery does it. The **QR code** toy makes a code from a link, plain text, a Wi-Fi network, a contact card, an email, a phone number, a text message, or a map location, and draws it as splats in one of seven styles (Classic, Dots, Rounded, Bricks, Gems, Bubbles, Neon) on a plate of paper, wood, or metal, or on none. The encoding itself is done by Project Nayuki's QR Code generator [20], a small MIT-licensed library that is vendored and loaded only when the toy opens. Splashery asks it for a code and then marks which modules belong to the finder patterns, to the alignment patterns, and to the rest, so each can be built and moved as a piece of its own. A tap bursts the code into its modules, which fall and fly back and land exactly on their grid again; the toy also has an assemble motion, a flip, and an "alive" color wave. Each module moves as a solid tile, and the alive hue waves keep the gray values a reader sees unchanged (the electric patterns brighten dark modules by a bounded amount), so every frame of it still scans.

It checks itself. **Scan view** turns the code flat and square to the camera, and after every change the toy renders that view and reads it back. The reader is the browser's own `BarcodeDetector` when it reads QR codes, and otherwise the vendored jsQR library (Apache-2.0), loaded the first time a check runs. The result, "It scans," means that one reader read that one drawing, not that every phone camera will; the text beside it says so. Nothing leaves the device: no shortener, no network.

Error correction. A QR code carries extra codewords (a codeword is 8 modules) made with a Reed–Solomon code [19]. There are four levels, L, M, Q, and H, which can lose about 7, 15, 25, and 30 percent of the codewords and still scan. That

percentage counts codewords, not squares, and the code is cut into blocks that are corrected separately, so damage to the three finder squares, or damage spread over many blocks, can defeat a code that has lost far less than its limit. The toy uses level M by default.

Why these numbers? Contrast, module size, and level M

OUR CHOICE, MEASURED The QR standard sets the codes; the limits for splat styles are Splashery's own.

A splat is not a printed square: it is a soft blob, and the blobs of two cells blend a little at their edges. So Splashery measured how much a code can take. The QR scan lab (October 3, 2026) made 3,768 pictures of the toy's codes the way a phone sees them, and read each with two different readers (jsQR and ZXing). Every style read 100 percent of the time at levels L and M. Contrast was reliable from 4 to 1 (a gray code at 4.0 to 1 read 96 to 98 percent of the time) and poor below 3 to 1. A module needs at least 4 pixels on screen, and 6 for the "depth" styles (Bricks, Gems, and Neon). A depth style at version 8 or more (a long text) dropped sharply, to 77 percent for a long text. Light-on-dark codes read on most phone cameras but not on some scanner apps. The toy turns these into a warning and a default; a measured limit is only as good as the readers it was measured with.

Where it lives: the comment and constants at the top of `src/packs/qr.js` (`eccFor` , `minContrast` , `MIN_MODULE_PX`).

Secrets. A Wi-Fi password is never in the toy's options, so never in a shared link or a saved scene: it lives in the page's memory while the page is open.



The QR code toy



A picture woven into a code

Both codes scan. The toys check their own pictures.

More QR toys, and other barcodes

- **How a QR code works** lights up each part of a real code, or steps through encoding your own text, from the mode bits to the chosen mask.
- **The Damage lab** lets you scratch a code, stick a label on it, tear or burn a corner, smudge it, or damage its splats (blur, shrink, grow, jitter, fade, drift the colors, curve, tilt, move). A meter reads the very picture on the stage after each change, and counts, block by block, the codewords lost against the ones error correction can fix. *Heal it* shows the code as a reader read it and lets Reed–Solomon decoding set it right, block by block, until it scans again.
- **Three codes in one** puts three QR codes in the red, green, and blue of one square, which Splashery's reader splits apart.
- **Picture QR** weaves a photo into a code as a halftone. The center of each module keeps its bit; the rest of the module carries the picture as dark and light dots by error diffusion, nudged only where a module would misread. The toy measures the contrast and reads the code back with jsQR at phone size and smaller, and offers the closest version that scans when one does not.
- **QR from real things** builds a code from dominoes that topple into place, marbles that roll into their modules, or tiles that flip over, and ends on a code that scans.
- **Other barcodes** draws Code 128, EAN-13, and UPC-A with Splashery's own code (check digits and quiet zones included), and Data Matrix and Aztec with ZXing for

JavaScript, a vendored library loaded only when the toy opens. A tap sweeps a scanner's red line across, and each bar lifts as it passes.

All of these are on the Studio shelf and behind the labs switch.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/codes.html#demo-qr

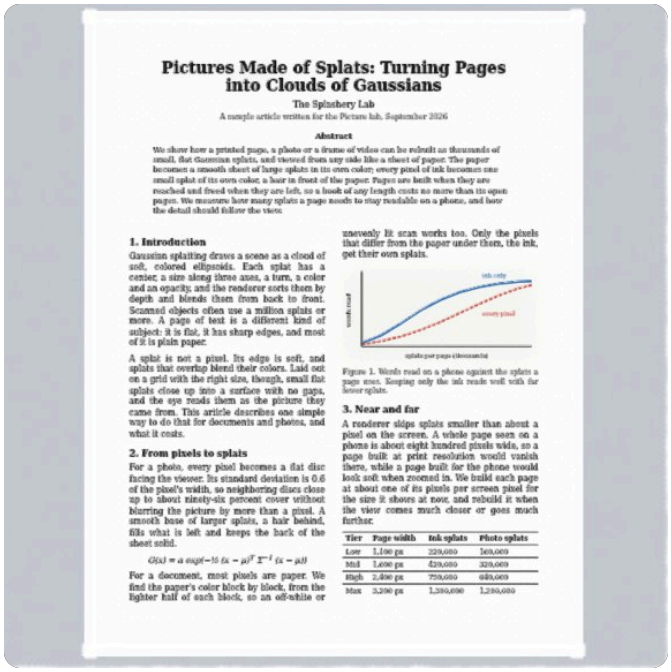
Pages, PDFs, and GIFs

What it is. Chapter 5 showed how a recipe places a *sheet*, a flat place where a picture shows, and how it turns a page. This section is about what the site does behind a sheet.

How Splashery does it. A PDF is drawn page by page by PDF.js [22], vendored and loaded only when someone opens a PDF. A worker turns each page's pixels into splats. Pages stream through a few sheets, so a 200-page book costs the same as its open pages, and a small cache keeps the pages next to the open ones ready. Each sheet is built at about one picture pixel per screen pixel for how big it shows now, on a ladder of widths a square root of two apart, within the device's budget, and is rebuilt when the view comes much closer or goes much further away: so a page you zoom into gets sharper, and a page you leave is freed. A GIF (decoded by omggif, vendored and loaded only for a GIF) or a video is a screen sheet: the splats are built once and take their colors from the sheet's own texture, which each new frame uploads. A PDF's words are available from its text layer, for reading, selecting,

copying, and finding, and a link in a PDF is tinted on the page; a link opens only after the site asks you.

The toys on top are **Picture lab**, **Your book** (a PDF as a book whose pages turn like paper, and wait until the next page is ready), **Photo album**, **Picture frame**, and **Screen**. They are behind the labs switch.



Pictures Made of Splats: Turning Pages into Clouds of Gaussians

The Splashery Lab

A sample article written for the Picture lab, September 2026

Abstract

We show how a printed page, a photo or a frame of video can be rebuilt as thousands of small, flat Gaussian splats, and viewed from any side like a sheet of paper. The paper becomes a smooth sheet of large splats in its own color; every pixel of ink becomes one small splat of its own color, a hair in front of the paper. Pages are built when they are reached and freed when they are left, so a book of any length costs no more than its open pages. We measure how many splats a page needs to stay readable on a phone, and how the detail should follow the view.

1. Introduction

Gaussian splatting draws a scene as a cloud of soft, colored ellipsoids. Each splat has a center, a size along three axes, a turn, a color and an opacity, and the renderer sorts them by depth and blends them from back to front. Scanned objects often use a million splats or more. A page of text is a different kind of subject: it is flat, it has sharp edges, and most of it is plain paper.

A splat is not a pixel. Its edge is soft, and splats that overlap blend their colors. Laid out on a grid with the right size, though, small flat splats close up into a surface with no gaps, and the eye reads them as the picture they came from. This article describes one simple way to do that for documents and photos, and what it costs.

2. From pixels to splats

For a photo, every pixel becomes a flat disc facing the viewer. Its standard deviation is 0.6 of the pixel's width, so neighboring discs close up to about ninety-six percent cover without blurring the picture by more than a pixel. A smooth base of larger splats, a hair behind, fills what is left and keeps the back of the sheet solid.

$$G(x) = a\,\exp(-\tfrac{1}{2}(x-\mu)^{T}\,\Sigma^{-1}\,(x-\mu))$$

For a document, most pixels are paper. We find the paper's color block by block, from the lighter half of each block, so an off-white or unevenly lit scan works too. Only the pixels that differ from the paper under them, the ink, get their own splats.

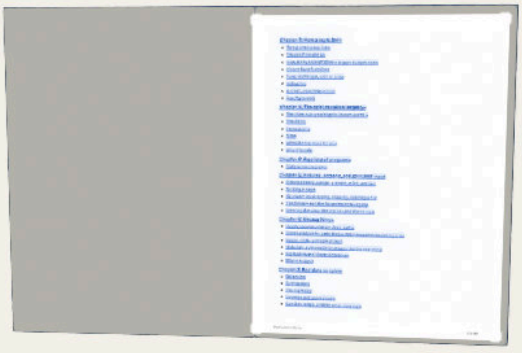
Figure 1. Words read on a phone against the splats a page uses. Keeping only the ink reads well with far fewer splats.

3. Near and far

A renderer skips splats smaller than about a pixel on the screen. A whole page seen on a phone is about eight hundred pixels wide, so a page built at print resolution would vanish there, while a page built for the phone would look soft when zoomed in. We build each page at about one of its pixels per screen pixel for the size it shows at now, and rebuild it when the view comes much closer or goes much further.

Tier	Page width	Ink splats	Photo splats
Low	1,100 px	220,000	360,000
Mid	1,600 px	420,000	320,000
High	2,400 px	750,000	680,000
Max	3,200 px	1,200,000	1,200,000

A PDF page, drawn with splats



Your book

The article is the toy's own sample text.

Save PDF goes the other way: from a toy to a page. With the labs switch on, the Share tab can write a PDF of the toy (a still, a link, and a QR code of the link), and, if you choose, a second page with a recording of one tap as a flip book, in which each frame is a hidden button whose look is the picture and the PDF's own JavaScript shows one at a time. A viewer without PDF JavaScript shows the first frame. The file is written by pdf-lib (MIT), loaded only when you export, and is saved to a file you choose.

A Wikipedia book. The rules in Chapter 7 allow a toy that fetches a Wikipedia article when the reader asks for it (its text is CC BY-SA, never stored in the repository, shipped with the site, or saved in a scene, and its credit shows beside it). Such a toy is planned. It is not built yet.

Share links

What it is. A Splashery link is not an address that points at your scene; it is your scene. It carries the toy's options, the look, the camera, and any paint, written into the web address after the `#`, which a browser never sends to a server. The scene is JSON, compressed with raw deflate (a `d.` link; a browser that cannot compress writes an uncompressed `j.` link instead) and written with the 64 letters, digits, `-`, and `_` that are safe in an address. A toy at its defaults is a link of about 700 characters; a link stops at 12,288 characters of scene, leaving out paint and then clay first. A link carries *settings, not files*: a photo or a splat file from your device has to be opened again by whoever opens the link.

Where it goes next. A link is text, so it can be a QR code. The share links chapter takes one apart, lets you break one, and counts how many characters a code can hold. Here, notice what codes and pages have in common with links: they all put a lot of meaning in a small, exact picture.

Where to try it

- QR code, How a QR code works, QR damage lab, Three QR codes in one, Picture QR, QR from real things, and Other barcodes (all labs).
- Picture lab, Your book, Photo album, and Screen (labs).

CHAPTER 10

Make your own

The last chapter of the ladder is a short one, because most of it is the earlier ones: you make your own toy by writing a recipe (Chapters 2 to 6), by typing an equation (Chapter 3), by turning something you already have into splats (Chapters 7 to 9), or by opening a file someone else made. This chapter gathers those ways in one place and says, for each, what it takes and what it cannot do. It ends with how to share what you made.

Five ways to make your own

Way	What you give it	What you get	Where
Roll one	A shape, a palette, a seed	A made-up splat object, the same every time you use that seed	Make tab, always on
Type one	An equation for *x*, *y*, *z*, and a color	A surface of splats, and a link to share it	Splat equation (Chapter 3), labs
Open one	A splat file (`.ply` , `.sog` , `.splat` , `.spz`)	The splat as a toy that Poke, Paint, Magnet, and the effects work on	Make tab, always on
Turn something into one	A photo, a video, a 3D model, a table, a sound, a PDF, a molecule, a scan	Splats made on your device	The Studio, labs (below)
Write one	A recipe in JavaScript	A toy with parts, controls, and a tap of its own	The source code (Chapters 2 to 6)

Roll one and open one

Roll one. The **Make a toy** section of the Make tab builds a splat object in your browser from a few choices: a shape (sphere, noise blob, torus, capsule, or knot), a palette, a seed, how many splats, how much the sizes jitter, how rough the surface is, and how much color noise there is. The generator is seeded and pure (`src/generators.js`), so the same seed and settings give the same toy, which is

why a link can rebuild it. Part of the splats fill the inside, so the slice effect has something to show.

Open one. Your own splat opens a PLY, SOG, SPLAT, or SPZ file (SPZ versions 1 to 3). KSPLAT and version 4 SPZ files are refused with a message that says to convert them first. You can also drop a file anywhere on the page. The file stays on your device, and nothing is uploaded. Poke, Paint, Magnet, every effect, the motions, the flag and pattern colors, and the Detail setting all work on it. What cannot work is a toy's own action (a lid that opens: a captured file has no lid) and Clay. And a file cannot go in a share link, because it never leaves your device: a link carries settings, not files (Chapter 9). To share the result, export a picture (PNG), a GIF, or a video.

The Studio

What it is. The Studio is a shelf of tools that turn other things into splats, on your device. Each takes a file you open, does its work on your device (some tools in a worker), and shows the result as the toy's splats. They are behind the labs switch. Nothing is uploaded, and the libraries a tool needs (a PDF reader, a depth model, a laz decoder) are loaded only when its toy opens.

Toy	You open	It makes	Chapter
Photo to 3D	A photo	The photo at its depths, as a Sharp view or as splats	8
Moving photo to 3D	A GIF or a short video	The clip played back in 3D	8
Video to 3D	A video of a still scene	A trained splat scene and its camera path (WebGPU)	8
Model to splats	A glTF, GLB, OBJ with its MTL, or STL file	The surface as flat discs, colored from the model	10 (below)
Splat toolkit	A splat file (or two, to compare)	The file's numbers, cropped, cleaned, shrunk, and saved in another format	10 (below)
Point clouds	A LAS, LAZ, PLY, XYZ, or PTS file	The points as splats, colored by height, intensity, or class	7

Toy	You open	It makes	Chapter
Volume viewer	A DICOM, NIfTI, TIFF, or raw volume	A block of splats you can cut through	7
Data in 3D	A CSV or TSV table	A scatter, bars, or a surface with labeled axes	7
Molecule viewer	A PDB, mmCIF, SDF, or XYZ file	Balls and sticks, space-filling atoms, a cartoon, or a surface	7
QR code	A link, text, or other contents	A code that checks that it scans	9
Your book	A PDF	A book whose pages turn	9
Song landscape	A song	A 3D field made from its spectrogram	(sound toys)



Model to splats



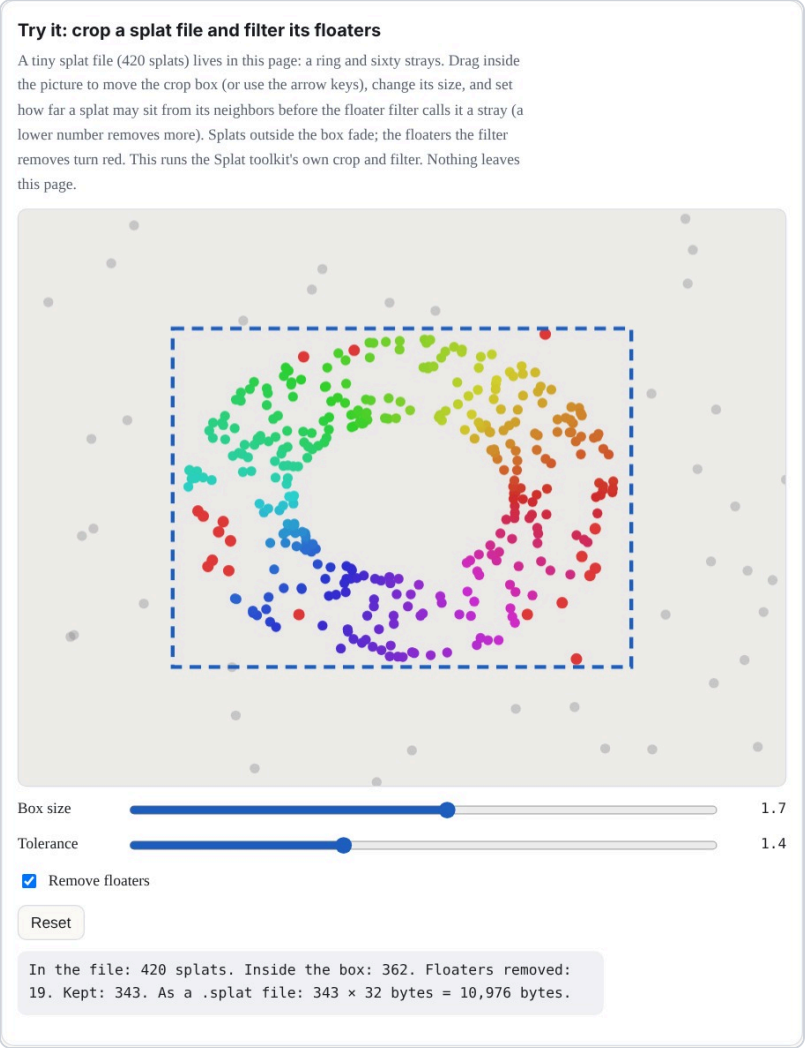
Splat toolkit

Models, and your splat files

Model to splats (`src/packs/studio-models-core.js` , which a Node tool shares) reads a glTF 2.0 file (`.gltf` or `.glb`), an OBJ with its MTL, or an STL file, and lays flat discs on the surface, more of them where there is detail, each sized

to its neighbors so the surface closes. It colors them from the model's base-color texture, vertex colors, or material, and the budget is the tier's default times 1.5, capped at its maximum, as for a photo. What it does not read: glTF compressed with Draco or meshopt, morph targets, skinning (a skinned mesh shows in its bind pose), animations, lights, cameras, emissive colors, and any texture except the base color. Glass, a material less than half opaque, is left out, because splats cannot be see-through solids.

The Splat toolkit opens a 3D Gaussian splat file and shows its numbers: how many splats, how big, how opaque, and the degree of their view-dependent color. Then it works on it: crop it with a box, remove floaters with a statistical filter you tune (the removed splats show in red, or before and after side by side), shrink it to a share of its splats, and compare it with a second file, turning in step. It saves in four formats: PLY (everything is kept; every tool reads it), SPZ (about a tenth of the size; view-dependent colors are kept), SOG (the smallest; base colors only), and `.splat` (simple; base colors only). A file is saved to a place you choose.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/make.html#demo-file

Writing a recipe

A recipe is a plain JavaScript object in a pack file under `src/packs/`, and the keys it can have are the whole of this manual so far. Here they are in one place:

Show the table: what a recipe can ask for

Key	What it is	Chapter
`build(k, o)`	Makes the splats from the kit's shapes, from the options `o`.	2
`drive(t, c, out, info)`	Every frame: moves parts, tokens, and levers; sets glow and sheets.	2, 5
`options`, `controls`, `action`	The choices saved in a link, the live sliders and pulses, and the tap.	2
`pictures`, `input`, `tiltLock`	What a visitor can open or type, and how a drag turns a flat toy.	5
`hands`, `handsOn`, `handsLevel1`	Pieces, joints, ropes, materials, and the ✋ switch. `handsOn: false` keeps a toy out of hands-on play; `handsOn: true` starts the switch on; `handsLevel1` lets a picture or a still toy play as one rigid body.	6
`alive`, `density`, `kernel`, `turntable`	The toy's own behaviors, how many splats it wants per base budget, the splat shape, and the idle spin.	2, 5
`credits`	The license and source notices that show beside the toy.	10

There is no editor in the browser yet; "What comes later" explains the plan. Today a recipe is a file in the project, and you can run your own copy of the site on your computer and look at a recipe as you write it. Chapter 2's *run it yourself* section shows how, with a prompt you can give to a coding assistant. A toy joins the shelf only after it has been checked and read by a person. Its `credits` name the source, author, and license of every asset it uses, and the site shows them beside the toy; an asset must carry a license the site can use (never one that forbids changes, and never one that is unlicensed or paid).

Sharing what you made

- **A link**: the Share tab copies a link that holds the whole scene. It works for everything except a file you opened (Chapter 9).
- **A scene file**: the same scene as a JSON file, which keeps paint and clay that a link is too short for. Old scenes (versions 2 and 3) keep opening.
- **A picture, a GIF, or a video**: PNG, GIF, and WebM exports of the view, and a live recording that starts when you tap Record and is saved only to a file you choose.
- **A page of a document** (labs): *Save PDF* (Chapter 9).
- **On a web page**: the Share tab gives an iframe snippet and a `<splashery-toy>` element, which takes a `scene` or a `toy`, a `theme`, a `background`, an `autoplay`, a `turntable` setting, a `zoom`, and a `label`. It starts when it scrolls into view and pauses when it leaves.

Every file you open stays on your device, and so does everything you make unless you choose to save it or share it.

Where to try it

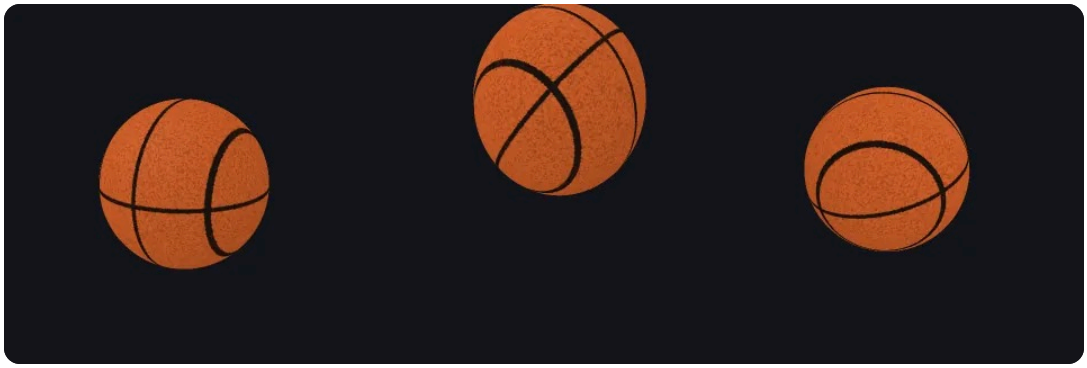
- Splat equation, Model to splats, and Splat toolkit (all labs). The Make tab in the Playground is open to everyone.

What else Splashery does now

This chapter lists what is on the site today, beyond the equation toy and picture toys you have read about. It describes only what is built. The **labs switch** hides newer parts until they are ready: add `?labs=1` to the site's address to turn it on (the browser remembers it; `?labs=0` turns it off). Each paragraph says what is behind it. The same list is a page on the site: What Splashery can do.

Hands-on physics

With the ✋ switch on, a drag picks up whatever is under your finger and a quick let-go tosses it. It lands, bounces, and settles on the floor under the toy, inside an invisible play area, and soft toys squish when they land. The ↺ button sends everything home. Most toys play this way, scans included, as one rigid body (`src/physics/hands-on.js`). A recipe can turn it off (`handsOn: false`), and picture toys and still toys play only if the recipe opts in (`handsLevel1`) or has joints. A recipe's `hands` block can instead make its pieces separate bodies, so a toy can have its own interactions. This is not behind the labs switch.



A ball at rest, just tossed, and in the air: three moments of one toss, drawn by the app.

The QR code toy

The QR code toy makes a code from a link, plain text, a Wi-Fi network, a contact card, or an email, and builds it from splats. Its motions (assemble, flip, burst, and return, and an "alive" color wave) move each piece as a solid tile, and the alive wave keeps the gray values a reader sees unchanged, so each frame still scans. A "Scan view" turns the code flat to the camera, and after every change the toy renders that view and reads it back, to check that it scans. The reader is the browser's own detector where there is one, and otherwise the vendored jsQR library; nothing leaves the device. A Wi-Fi password is not kept in a link. This toy is behind the labs switch.



The QR code toy, drawn from splats.



The QR damage lab as it opens (left) and Picture QR, a painting woven into a code (right).

Captures and file formats

The viewer opens `.ply`, `.sog`, `.splat`, and `.spz` files (version 4 SPZ files, which need a decoder Splashery does not ship, are refused with a message). Files can carry spherical harmonics, the view-dependent color the kit's splats do not use. The Scans shelf holds captures with their licenses and credits. The newer captures each ship as a full file and a lite file, and some keep a band of spherical harmonics on the full file. Many of the newer captures are behind the labs switch.



Two captures from the Scans shelf: a marble bust and a wooden elephant.



Two more captures: a bonsai tree and a sunflower.

Camera, microphone, and screen

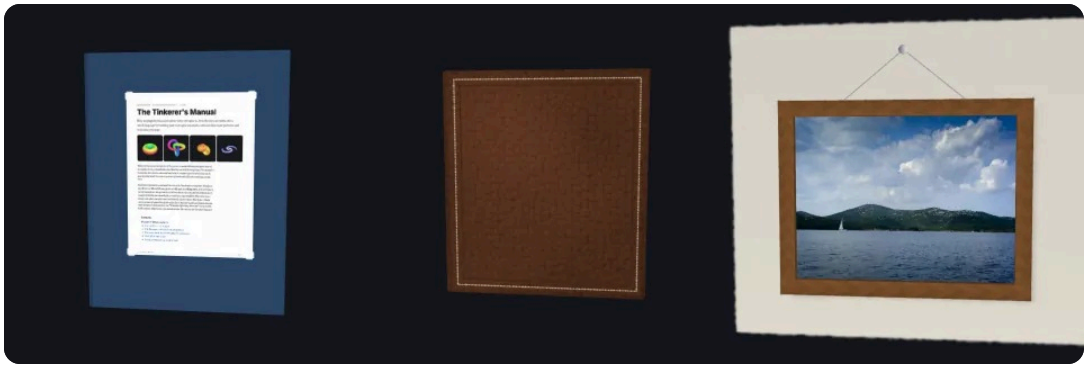
A toy can read the microphone, the camera, or a shared screen, but only after you tap a button for it. Nothing is requested before that tap. While a source is on, a small "Live" pill with a Stop button shows at the top of the page, and the sound or pictures stay on your device: nothing is sent anywhere, and nothing is recorded unless you ask. (One exception: a recorder, such as the splat mirror's, records only after you tap Record, keeps the recording on your device, and saves it only to a file you choose.) Two examples are the room echo meter (clap, and it measures how long the room rings) and the splat mirror (your camera as splats, with a depth model of about 27 MB that loads only after you start the camera). Both are behind the labs switch.



The room echo meter waiting for a clap (left) and the splat mirror as it opens (right).

Books, albums, and frames

Your book turns the pages of a PDF as paper: a page is a leaf with a front and a back, the engine can re-sort the splats of a turning page where it stands, and a turn waits until the next page is ready so it never shows a blank one. The words of a PDF page are available from its text layer, and links in the PDF are tinted on the page. The photo album opens several photos at once, and the picture frame lets you list them and move each up or down. These toys are behind the labs switch.



A book, a photo album, and a picture frame.

Studio: sound and photos

The Studio shelf turns other things into splats on your device. The song landscape makes a spectrogram of a song you open into a 3D field, and the Chladni plate shows sand finding a plate's still lines. Photo to 3D runs a depth model on your device, works out how far each part of a photo is, and rebuilds the photo as splats at their depths. The Studio toys are behind the labs switch.



The Chladni plate with its sand on the still lines (left) and a photo rebuilt as splats (right).

Models to splats

The Model to splats toy opens a glTF or GLB file, an OBJ with its MTL, or an STL file and turns it into splats on your device: flat discs on the surface, more of them where there is detail, each sized to its neighbors so the surface closes, and colored from the model's texture, vertex colors, or material. A toy's input can take several files at once, for a glTF with its `.bin` and textures. This toy is behind the labs switch.



The Model to splats toy as it opens, with its sample model drawn as splats.

Worlds

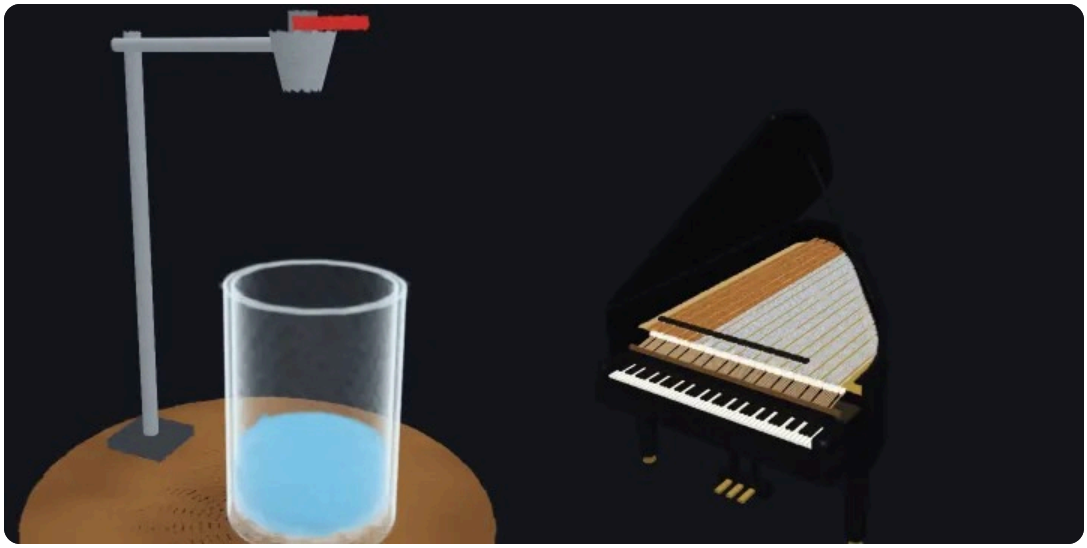
A world is a small place you walk around in, built on your device from recipes and a seed. It has a start screen, a list of places, collision, a character, and a camera that follows. In splats mode everything you see is splats. In hybrid mode the ground, water, sky, and signs are lit models, and only the props and the near grass stay splats, so "everything is splats" holds for the toys but not for every part of every world. Worlds are behind the labs switch (`worlds/?labs=1`).



A small world you walk around in.

Fluids, songs, and smooth bodies

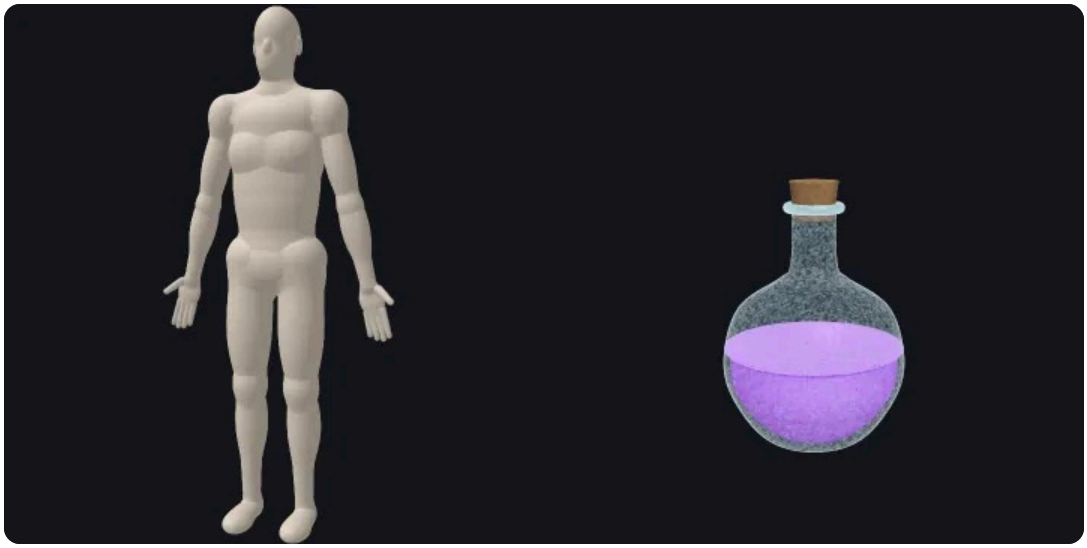
A recipe can declare a liquid, a gas or a flame with `k.fluid`. It is simulated while the toy is open, and the engine loads only when such a toy opens. How it is drawn depends on the fluid and on the device. On the page's own path (`src/fluids/render.js`) each liquid particle is four small splats. On a device with WebGPU, the liquids run on the graphics chip instead and are drawn as one smooth surface, in a pass after the splats (`src/fluids/gpu/surface.js`). Smoke, steam, and flames run on a grid on either kind of device and are drawn as a glowing volume (`src/fluids/gpu/gas.js`); where the grid is off, their particles are splats. So "made of splats" holds for the simplest path, but it is not the only way a fluid is drawn. A keyboard toy can play songs from its own list, a MIDI file, or a tune pasted as ABC notation, with every key, hammer, and string moving as a solid piece. Recipes can also join shapes into one smooth body and place one toy's recipe inside another's (the anatomy atlas does both). The Fluid lab, the bottle toys, and the anatomy atlas are behind the labs switch.



The Fluid lab (left) and a grand piano that plays songs (right).



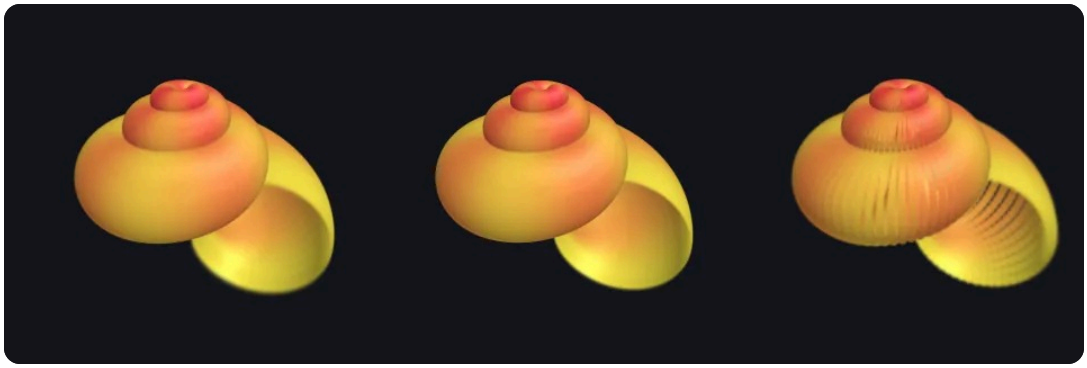
Flames and smoke: a campfire (left) and a candle (right).



The anatomy atlas (left) and a bottle toy, the potion bottle (right).

Solid, Fine, and Dots, and the detail you see

The Splat equation toy has three looks. **Solid** lays a surface's splats flat along it so they close into a clean shape, **Fine** uses up to four times as many smaller splats within the device's budget, and **Dots** shows each splat you programmed as its own round dot. The toy also asks for the sharper labs kernel. Beyond that toy, the viewer's Detail setting (Auto, High, or Max, and with labs on a slider from 60,000 to 400,000 splats) lives in your browser only, never in a link, and on every toy the canvas resolution drops while you drag if frames are slow. Those two are not behind the labs switch; the toy and the sharper kernel are.

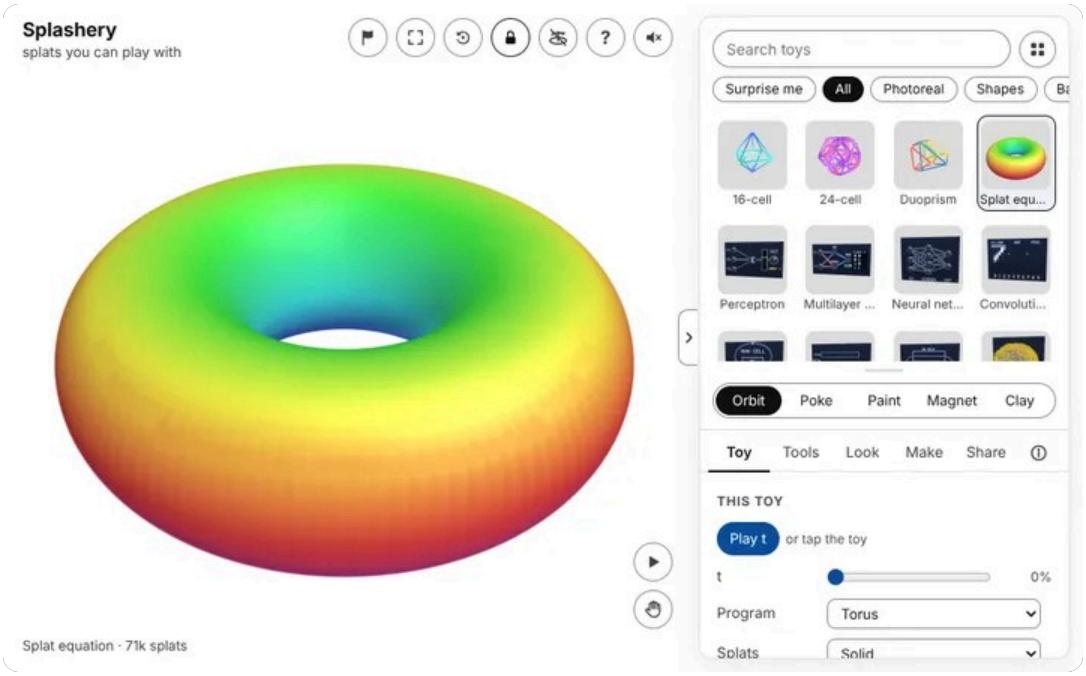


One program, three looks: Solid, Fine, and Dots.

The viewer's controls

Focus mode hides everything but the toy (the **F** key or the focus button, and **Esc** brings the rest back). On a computer, **[** folds the side panel and **G** opens the gallery.

A toy's input panel can take a typed text, a drawing, a file, a binary file such as a sound, or several files at once. These controls are for everyone.



The viewer: the toy on the left, the shelf and the panel of its controls on the right.

What Splashery can do that other splat tools can't

PlayCanvas is the engine that draws the splats, and its editor, SuperSplat, works on captured splat files: you crop, clean, and arrange a capture. Splashery starts from the other end, because most of its toys are never captured at all. These are things we have not seen in other public splat tools. (We have not tried every tool, so read this as "to our knowledge".) The site has this list as a page too: What Splashery can do.

- **Toys made from rules, in the browser.** A recipe of a few kilobytes is built into splats on your device. Its base budget is 60,000 to 400,000 splats, depending on the device and, in labs, the Detail slider; a recipe that asks for more density gets more, up to the tier's most, so those are the budget a toy is asked for, not a count every toy draws. The shape itself is never a multi-megabyte file to download. (The page, the engine, and any optional models or captures a toy uses still download, and only when that toy opens.)



A cactus, a strawberry, and a knot: three toys built from rules, not downloaded as shapes.

- **Equations as a language.** Type *x*, *y*, and *z* in *u*, *v*, and *t*, and get an animated splat shape, with plain messages instead of crashes (Chapter 3).



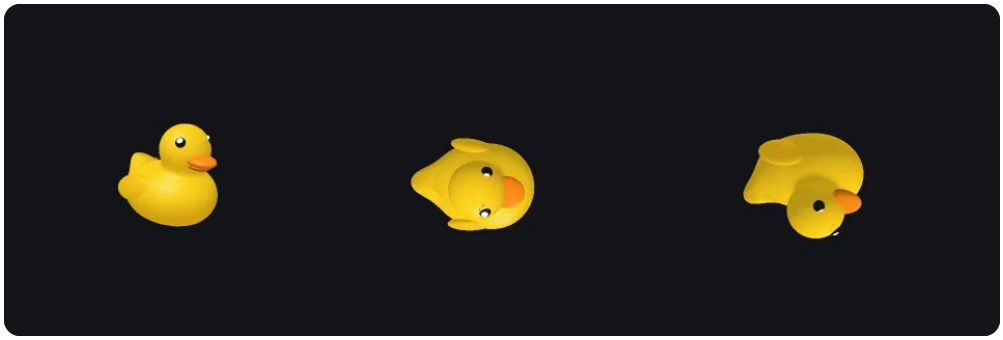
Three programs of the equation toy: a Klein bottle, a trefoil knot, and a galaxy.

- **Moving parts inside one splat cloud.** Hinges, pivots, sliders, morphs, and per-splat effects run on the graphics chip, so a chest opens and a windmill turns without rebuilding anything (Chapter 2).



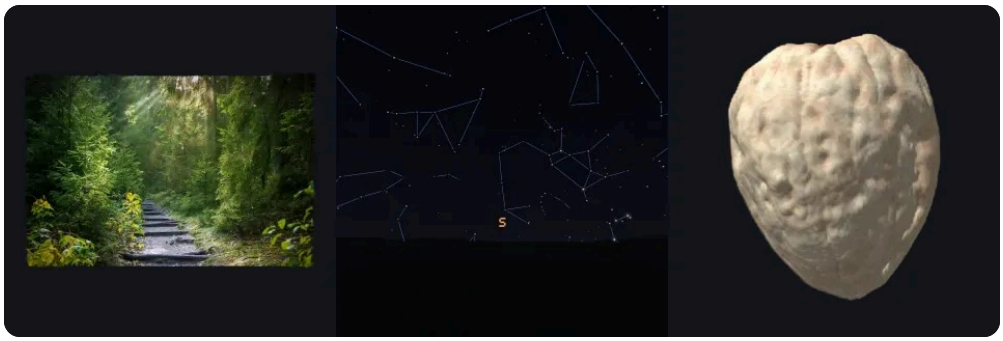
A chest opening on a hinge: one splat cloud, three moments.

- **Hands-on physics on splats.** Toss, stack, pull apart, and pour.



A rubber duck at rest, tossed, and tumbling.

- **Anything as splats.** Photos and videos with depth, PDFs, QR codes, real science data, the night sky, and medical volumes.



A photo, the night sky, and a scanned volume, all as splats.

- **A whole scene in a link.** The `#s=` links carry the toys, their options, and the camera. How they work:
  - The scene is written as JSON, compressed with raw deflate, and written in base64url, with a `d.` in front (`src/codec.js`). Where a browser cannot compress, the link carries `j.` and the same JSON, uncompressed but still in

base64url, so it is not readable as plain text. Share links and QR codes explains the whole path.

- A link carries settings, not your files. A photo or a splat file from your device has to be opened again by whoever opens the link.
- No server is involved. The link is the whole scene.

The scene, as JSON

137 bytes

```
{"app":"splashery","version":3,"seed":7,"toy":{"kind":"bu…
```

Squeezed (raw deflate)

108 bytes

```
the same facts, said in fewer bytes
```

Written in 64 safe letters (base64url)

144 characters

```
d.FcpBCsMgEAXQu_z1EAKpFOY2Ux…
```

The link

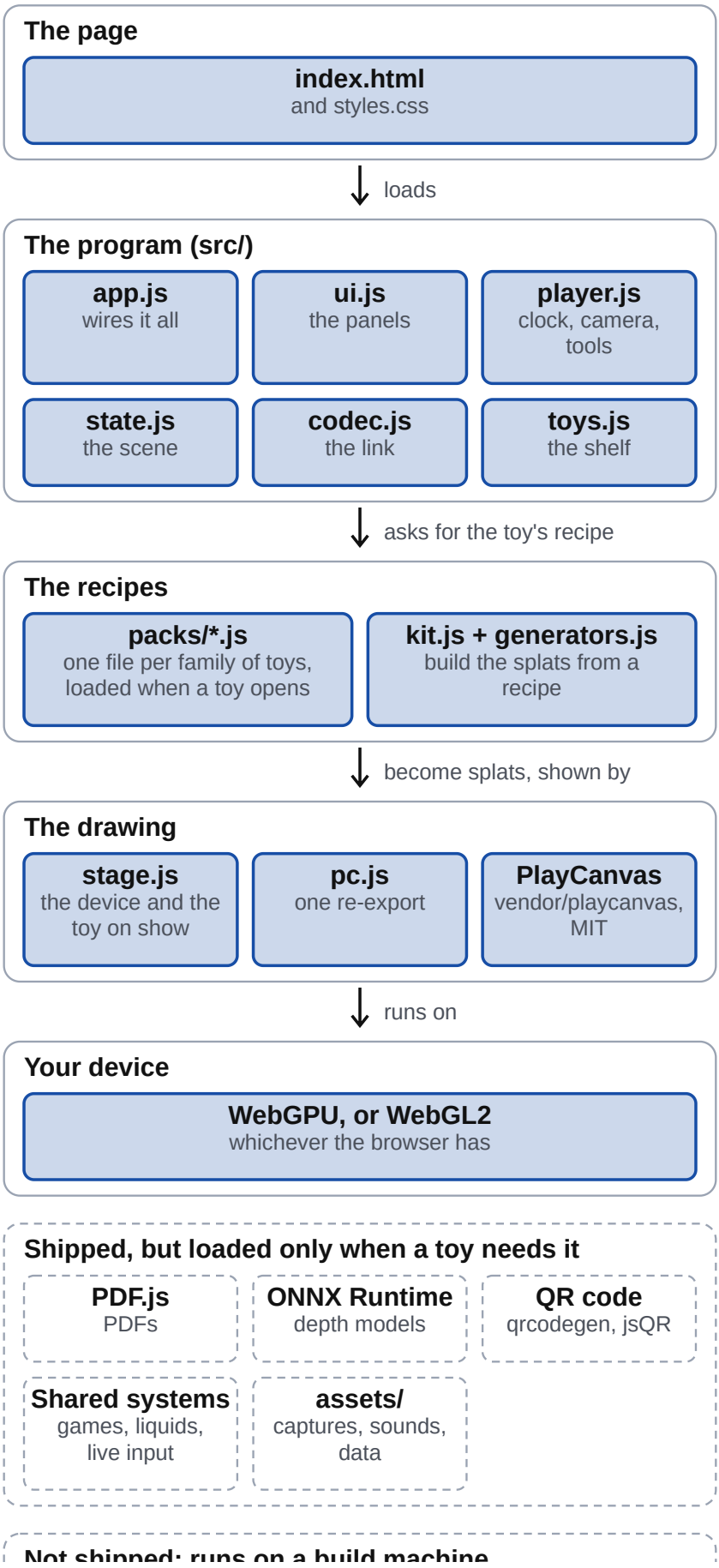
146 characters after #s=

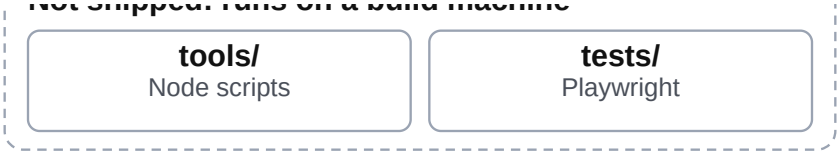
```
…/splashery/#s=d.FcpBCsMgEAXQu_z1…
```

A tiny scene (a cactus and a camera), taken through the real link code, stage by stage.

How Splashery is built

Everything else in this manual is about what a toy is. This chapter is about the program that holds the toys: where its files are, why it is made of plain files with no build step, what the PlayCanvas engine does for us, what we wrote ourselves, and every outside library that ships with the site. The file names are real. Open the source beside this page and you can follow along.


The page
index.html
and styles.css
loads
The program (src/)
app.js
wires it all
ui.js
the panels
player.js
clock, camera, tools
state.js
the scene
codec.js
the link
toys.js
the shelf
asks for the toy's recipe
The recipes
packs/*.js
one file per family of toys, loaded when a toy opens
kit.js + generators.js
build the splats from a recipe
become splats, shown by
The drawing
stage.js
the device and the toy on show
pc.js
one re-export
PlayCanvas
vendor/playcanvas, MIT
runs on
Your device
WebGPU, or WebGL2
whichever the browser has
Shipped, but loaded only when a toy needs it
PDF.js
PDFs
ONNX Runtime
depth models
QR code
qrcodegen, jsQR
Shared systems
games, liquids, live input
assets/
captures, sounds, data
Not shipped: runs on a build machine



The parts of Splashery, top to bottom: what the browser loads, and what draws the toy.

Plain files, and nothing else

Splashery is **static files and ES modules**. There is no bundler (nothing turns many files into one), no server program, no database, no content delivery network, and no key to a paid service. A web page asks for `index.html`; that page asks for `src/app.js`; and the browser follows each `import` line to the next file, one request each. The site is served as it sits in the repository (GitHub Pages serves the `main` branch), and to run it on your own computer you start any file server in the folder, for example `python3 -m http.server 4173 --bind 127.0.0.1`. There is nothing to install first, and `package.json` has no build step.

Why this way? No bundler, no server

OUR CHOICE It is a ground rule of the project (`CLAUDE.md`), kept on purpose. Four things follow from it. A toy can be **read**: the file you open in the browser's tools is the file in the repository, not a squeezed copy. The site **costs nothing to run** beyond free file hosting, and has no application server of its own that could fail. (The files still have to be served by some host, GitHub Pages today, or a file server on your own computer, and the page loads them over the network, so it can still be unreachable when the host or the network is down.) Your scenes and files **stay on your device**, because there is no server to send them to (the Terms of use in the About tab say so). And a page that embeds a toy (`embed/index.html` , the `<splashery-toy>` element in `src/element.js`) needs only a few static files, with a test that keeps an embed under 30 MB.

If it were chosen differently. A bundler would make the first load smaller and faster (one file instead of hundreds, with unused code removed). A server would allow short links, saved accounts, and heavy work done off the phone. Each costs something the four points above give for free. The cost of the choice today is that a rarely used part must be loaded by hand, only when its toy opens (see the libraries below).

Where it lives: `index.html` , `package.json` , `playwright.config.mjs` (the test server is the same file server), and `CLAUDE.md` (the ground rules).

Where the files are

Folder or file	What is in it
`index.html`, `styles.css`	The app: one page, its styles, and a small import map for the GIF encoder.
`src/`	The program. The core is a few files: `app.js` and `ui.js` (wiring and panels), `player.js` (the clock, camera, effects, and tools for the toy on show), `stage.js` (the one place that talks to the engine), `state.js` and `codec.js` (the scene document and the link; see "Share links and QR codes"), `toys.js` (the shelf), and `kit.js` with `generators.js` (the toy kit, Chapter 2).
`src/packs/`	The toys' recipes, one file for each family of toys (about a hundred files): `balls.js`, `food.js`, `splat-equation.js` (the language of Chapter 3), and so on. A recipe is the JavaScript of Chapter 2; `src/toys.js` lists each toy with the pack it comes from and whether it is behind the labs switch.
`src/arcade/`, `src/fluids/`, `src/physics/`, `src/qr/`, `src/live/`, and other folders	Bigger systems that more than one toy shares, each in its own folder: the games, the liquids and flames, the hands-on physics, QR codes, and the microphone and camera. Most are loaded only when a toy that needs them opens.
`assets/`	Captured toys (as `.sog` files), thumbnails, flags, and the sounds and data some toys read. Every asset's license is in `CREDITS.md`.
`vendor/`	Other people's libraries, copied in as they were published. Four needed a small change to load here (a comment removed, or a line added so the file loads as an ES module, or the TypeScript compiled to JavaScript); `LICENSES.md` records each one. The next two sections list them all.
`tools/`	Node scripts that run on a build machine and are never loaded by the site: they prepare assets, make thumbnails and clips, build the site's pages (`tools/site-build.mjs`), print this manual's PDF (`tools/manual-pdf.mjs`), and keep the plan of every toy (`tools/toy-plan.json`).

Folder or file	What is in it
`tests/`	The Playwright tests, which open the real site in a real browser. For example, `tests/taps.spec.mjs` finds every toy's tap by itself, so a new toy is tried the day it is added.
`embed/`, `worlds/`, `showcase/`, `site/`, `manual/`, `pdf-lab/`	The other pages of the site: the embed player, the Worlds, the Showcase, the guides and Learn hub (`site/`), this manual, and the PDF lab.

What PlayCanvas does, and what is our own

PlayCanvas is an open source (MIT) web graphics engine. Splashery vendors version 2.22.3 as one file, `vendor/playcanvas/playcanvas.min.mjs`, and every other file reaches it only through `src/pc.js`, which does nothing but re-export it. The ordinary toys never touch it: `src/stage.js` (with `src/paint.js` and `src/loaders.js`) is the one place that sets up the device and shows a toy, and only about two dozen files in all (the stage, the fluids, the Worlds, the live relief, and a few toys that draw extra layers) import `src/pc.js`.

PlayCanvas does	Our own code does
Opens the graphics device: WebGPU when the browser has a working one, otherwise WebGL2 (`deviceOrder` in `src/stage.js` picks).	Says which to try first, shows a plain message when neither exists, and falls back to a poster.
Holds the splats in textures, projects each one to the screen (Chapter 1), draws it as a quad with a Gaussian falloff, and blends them back to front.	Decides which splats exist: the kit and generators build them from recipes (`src/kit.js`, `src/generators.js`), and the equation language makes them from your formulas.
Sorts the splats by depth on a worker, a few frames after anything moves (the comment in `src/stage.js` about "splat sorting" is about this).	Keeps frames coming while the sort finishes, and asks for a new sort when the toy moves.

PlayCanvas does	Our own code does
Reads `.sog` and `.ply` splat files and runs a function on every splat each frame (a "work buffer modifier").	The modifier itself: the per-splat effects and the kit's parts, hinges, morphs, and colors, written in two shader languages (`src/effects.js`), plus the sharper falloff in `src/kernels.js` . Also reads `.splat` and `.spz` files (`src/loaders.js`).
The scene graph, a camera, and a loop that calls us each frame.	The orbit camera and touch gestures (`src/camera.js`), paint (`src/paint.js`), picking, sound (`src/sound.js` , WebAudio), the panels, the scene document, and every toy.

So the engine is the drawing machine, and the toys are ours. Splashery is **primarily splat based**, not entirely: the photo toys' Sharp view is a textured mesh of triangles (`src/live/relief-mesh.js`), the WebGPU liquid surface and the gas grid are extra passes drawn after the splats (`src/fluids/gpu/`), and the hands-on physics and the games' rules are ordinary JavaScript with no drawing in them at all. (See "What else Splashery does now".)

Where it lives: `src/pc.js` (the one import), `src/stage.js` (the device, the toy's entity, the sort), `src/effects.js` and `src/kernels.js` (the shader code), `vendor/playcanvas/LICENSE` .

Every library that ships, and when it loads

These are all the outside libraries and models in `vendor/` , the only files in the site that we did not write. `LICENSES.md` has the full notice and source for each. The rule (see `CLAUDE.md`) is that a library is vendored, loaded only when the toy that needs it opens, and listed there before it is used. Nothing is fetched from a content delivery network or an outside server, and no key is needed.

Library (folder in `vendor/`)	Its job	License	When it loads
PlayCanvas 2.22.3 (`playcanvas`)	The drawing engine, above.	MIT	With every page that draws a toy (the app, the embed, the Worlds).

Library (folder in `vendor/`)	Its job	License	When it loads
gifenc 1.0.3 (`gifenc`)	Writes an animated GIF when you export one.	MIT	With the app (`src/exports.js` imports it through the import map in `index.html`); it is small.
PDF.js 6.3.289 (`pdfjs`)	Reads a PDF so its pages can become splats.	Apache-2.0 (its image decoders: BSD and MIT)	Only when you open a PDF (`src/media.js`).
omggif 1.0.10 (`omggif`)	Reads a GIF's frames in browsers that cannot decode them (Safari).	MIT	Only when you open a GIF in such a browser.
ONNX Runtime Web 1.30.0 (`onnxruntime-web`)	Runs a neural network on your device, in WebAssembly.	MIT	Only when you open a photo or start a camera for depth, or a recording in Note Rider.
Depth Anything V2 Small, int8 (`depth-anything-v2-small`, about 27 MB)	The model that guesses how far away each part of a photo is.	Apache-2.0	Only when a toy needs depth (Photo to 3D and its relatives).
Basic Pitch model (`basic-pitch`, about 230 KB)	The model that finds the notes in a recording.	Apache-2.0	Only when you open a recording as your own song in Note Rider.
Splat.js 0.1.0 (`splatjs`)	Trains a splat scene from the frames of a video.	MIT	Only when you open a video in Video to 3D (labs).
Project Nayuki's QR Code generator (`qrcodegen`)	Writes the QR code's modules.	MIT	Only when something needs a code drawn: the QR code toy, the Share tab's scene code, or Save PDF (labs).

Library (folder in `vendor/`)	Its job	License	When it loads
jsQR 1.4.0 (`jsqr`)	Reads a QR code back, to check that it scans.	Apache-2.0	Only when a code is read back and the browser's own barcode reader is not used or cannot read it. The QR code toy asks for jsQR directly, and so does the check that Save PDF runs on its code; the Share tab's scene code uses the browser's reader first and falls back to jsQR.
ZXing for JavaScript 0.21.3 (`zxing-js`)	Writes and reads Data Matrix and Aztec codes.	Apache-2.0	Only when the "Other barcodes" toy opens (labs).
pdf-lib 1.17.1 (`pdf-lib`)	Writes a PDF when you choose Save PDF.	MIT	Only when you save a PDF from the Share tab (labs).
laz-perf 0.0.7 (`laz-perf`)	Reads a `.laz` point cloud.	Apache-2.0	Only when you open a `.laz` file in Point clouds (labs).
dicom-parser 1.8.21 (`dicom-parser`)	Reads a DICOM medical scan.	MIT	Only when you open a DICOM file in the Volume viewer (labs).
libwebp, from splat-transform 3.6.1 (`webp`)	Reads and writes the lossless pictures inside a `.sog` file.	MIT and BSD-3-Clause	Only when you open or save a `.sog` in the Splat toolkit (labs).
Fredoka, bold, Latin subset (`fredoka`, about 15 KB)	The rounded letters of "Your name in splats", as in the Toy Box name toy.	SIL Open Font License 1.1	Only when "Your name in splats" opens (labs).

The earthquakes toy and the molecule viewer's fetch by Protein Data Bank code do read from outside the site while you use it, but they fetch data (a feed, a structure), not code, only when you ask, and a dated snapshot of the earthquake feed ships for when it cannot be reached. The white stork migration is not one of them: it reads a

data file that ships with the site. The outside download of that data happens earlier, when a tool in `tools/` builds the file on a build machine, not in your browser.

> **What is used to build the site, but is not in it**
>
> `package.json` lists pinned `devDependencies` (the Playwright test runner, Prettier, a splat converter, glTF readers, and the QR readers used to test codes). Some programs outside npm (Blender, the Brush and msplat trainers) make the captured splats on a build machine. None of them is served; `LICENSES.md`, under "Development tools" and "Build tools outside npm", lists each with its license.

How a toy gets on the shelf

Putting this together: a toy is a recipe in a file in `src/packs/`, one entry in `src/toys.js` (its id, label, category, pack, and whether it is labs), its sound in `src/toy-sounds.js`, its help text in `src/toy-help.js`, and its plan in `tools/toy-plan.json`. Nothing is compiled or registered anywhere else. The next time the page loads, the toy is on the shelf, and the tests find it.

Where it lives: `README.md` ("Layout"), `LICENSES.md`, `CREDITS.md`, `docs/PACKS.md` (how to write a recipe).

Share links and QR codes

A Splashery link does not point at your scene. It *is* your scene. Open the Share tab, copy the link, and the whole toy (its options, the look, the camera, any paint) is written into the web address itself. This chapter takes one apart, lets you make one, and shows how the same link fits in a QR code.

What is in a link

The link has four parts, from left to right:

`https://ryanjosephkamp.github.io/splashery/` **`#s=d.`** `eJxVjkEOgzAM` **`…`**

1. **The page.** The ordinary address of the app. The browser asks the server for this part and nothing more.
2. **`#s=`**, the start of the *fragment*. A browser never sends the fragment (everything after the `#`) to a server, so the scene is not in any request. The page reads it with script after it loads (`parseHash` in `src/codec.js`).
3. **A format letter and a dot:** `d.` means the rest is compressed, and `j.` means it is not (the next paragraph says when).
4. **The scene.** The scene as JSON text (its shape is in `docs/SCENE-SCHEMA.md`), turned into bytes as UTF-8, **compressed with raw deflate** (the same method as a zip file, without the zip's wrapper), and written with the 64 letters, digits, `-` and `_` that are safe in an address (**base64url**).

The `j.` form exists for a browser that cannot compress (no `CompressionStream`): it is the same JSON, written in base64url but not compressed. It is not readable as text, only safely spelled. Any browser reads either form if it can decompress, and every current browser can.

It is not a hash of anything, and not a lookup. Some sites make a short link by saving your data on a server and handing you a key. Splashery cannot do that, because it has no server (the previous chapter). Nothing is stored anywhere. The link is the only copy of the scene, which is why you can send it in a message and the other person sees exactly your toy, and why a scene you did not copy the link of is not recoverable from the site.



Interactive in the web version: ryanjosephkamp.github.io/splashery/manual/share.html#demo-share

Why one changed character usually breaks it

Compressed text is not like ordinary text. In a sentence, a wrong letter is one wrong letter. In a deflate stream each bit tells the decoder how to read the bits after it (which code table, how far back to copy from), so one changed character changes how everything after it is read. The decoder then either stops with an error (`decodeSceneHash` throws, and the app shows "Could not read the shared scene"), or produces text that is no longer valid JSON, which stops at `JSON.parse`. Rarely, the changed character lands on a spare bit at the very end or turns one letter inside a word of the JSON into another, and the link still works with a slightly different scene. That is the "usually". Do not edit a link by hand to change a setting. Change the setting in the app and copy the new link.

When a link does decode, the app does not trust it. The JSON goes through `normalizeScene` (`src/state.js`), which drops keys it does not know, clamps every number to its allowed range, and fills in anything missing with its default. The demo shows the result of that step below the decoded text.

Old links keep loading

A link made last month must open next year. The scene has a `version` number, and `normalizeScene` reads old ones. Version 3 is the current schema. A **version 2** link is a version 3 scene without the pattern, the motion, and the options on shelf toys; it loads, and those take their defaults. A version 1 file (the old planet painter) is refused with a message that says so, and so is a scene from a newer Splashery than the one you are running. Saving always writes the current version. The rule is written in `CLAUDE.md` and is tested.

The same applies to the embed player and the `<splashery-toy>` element, which read the same links (`src/viewer.js` , `src/element.js`).

How long a link can be

A toy at its defaults is a link of about 700 characters. Paint and clay are the things that grow, because every dab or stroke is written into the scene. Sharing stops at 12,288 characters of scene (`LINK_LIMIT` in `src/exports.js`). Past that, `buildShareHash` first leaves out the paint, then the clay, says so, and if the link is still too long it tells you to save the scene as a JSON file instead. A link carries **settings, not your files**: a photo or splat file from your device has to be opened again by whoever opens the link.

A scene as a QR code

A QR code is a picture of bytes, and the link is just text, so any link can be one. With labs on, the Share tab has a **QR code** button (`src/qr/share.js`). It builds the same link as the Copy button and draws it as a QR code you can scan with a phone camera and save as a picture. The code holds *only the link*. Nothing is uploaded, and your own files stay on your device exactly as they do for a link.

There is a ceiling. A QR code at its largest (version 40, the lowest error correction) holds 2,953 bytes, and the whole link counts, address and all. A code carries extra codewords (a codeword is 8 squares) from which a scanner can rebuild the data when some of it cannot be read, and the more of them it carries the fewer bytes it holds. Level M can rebuild about 15 percent of the codewords and level L about 7 percent. That is a rate for codewords, not a promise about any 15 percent of the squares: damage that covers the three big squares that locate the code, or lands on many blocks of codewords at once, can defeat a code that lost less, while many damaged squares inside one codeword cost only that codeword. `shareCode` (`src/qr/share-code.js`) picks level **M** while the code stays at version 30 or

smaller, and level **L** beyond that, for the smallest code a phone can still resolve. In the measurements of the QR lane, every toy at its defaults came to about 690 to 715 characters. A link of up to 711 characters fits version 21 at level M, and one of 712 to 779 needs version 22, so most toys are version 21 and a few, depending on the scene's random seed, are version 22. A toy with 20 dabs of paint was about 1,170 characters (version 28), 60 dabs was about 2,000 (version 33, level L), and 150 dabs was 3,671, too long for any code, so the dialog says so and offers Save scene. Nothing is split across several codes.

The dialog shows the code at once, and under it says "Checking that it scans…" while it reads the code back (with the browser's own barcode reader, or `vendor/jsqr` where that cannot read QR codes). It changes that line to "It scans" only if the text it read back is exactly the link; otherwise it says the code did not read back and suggests Save scene. That is one reader on one drawing on this device, not a promise about every phone camera. The demo above does the same sizing with the real `shareCode`: it shows the version and level of its own link, and draws it.

Where it lives: `src/codec.js` (encode, decode, `parseHash`), `src/state.js` (`normalizeScene`), `src/exports.js` (`buildShareHash`, `shareURL`, `LINK_LIMIT`), `src/qr/share.js` and `src/qr/share-code.js` (the QR code), `docs/SCENE-SCHEMA.md`, and the tests `tests/qr4-2.spec.mjs` and `tests/man5.spec.mjs`.

Open questions: other choices, not tried yet

Splashery works as it is, and nothing here would replace what works. Each of these would be a separate experiment, behind the labs switch or on a comparison page, with today's behavior as the default. If you wonder "why this number?", the answer is in the notes in Chapters 1 to 3, or it is "not tested yet" and listed here.

Show the table: every choice not tried yet

Choice	What an experiment would show	Status
A Detail slider	Any splat count from 60,000 to 400,000, beside the tiers. Built as a labs slider (`docs/lab/DETAIL-SLIDER.md`).	Built, in labs
View-dependent color	A shine that moves with the view on a few kit toys, from the kit's own data and shader. Built as a private comparison, on only when a link or the address asks for it with labs on (`docs/lab/GLOSS.md`).	Built, in labs
More moments of *t*	Turning shapes at more than 12 moments, up to 15. Built as a labs choice (`docs/lab/MOMENTS.md`); more than 15 is still open.	Built, in labs
240-character fields	Longer fields, and what longer links cost. Built in the Splat equation toy (`docs/lab/EQUATION-FIELDS.md`).	Built, in labs
A budget sweep on real phones	Real phone graphics chips, and more phone sizes. The Kit lab's sweep (Chapter 2) covered five budgets, one phone size, and one software renderer.	Not started
More than 15 parts	A toy with 24 or 32 moving parts. The largest change: it repacks the part number, moves the part table into a texture, and needs a check on real phones.	Open: no plan yet
An opacity field	See-through splats in a program (and the other candidate fields in Chapter 3).	Open: no plan yet
Formulas on the graphics chip	Formulas compiled to a shader, so *t* is continuous and no copies are stored.	Open: no plan yet

What comes later: Code a toy

The splat equation language can place and color splats, but it can't give a toy parts, controls, or a tap of its own. For that you need a full recipe (Chapter 2). Splashery's plan includes a **Toy Workshop** where you will be able to write one: a **Code a toy** editor that builds and shows your recipe as you type.

Sharing those toys will work differently from sharing equations. A recipe is real JavaScript, and a page that ran a stranger's JavaScript straight from a link could be made to do things its visitor never agreed to. So a coded toy will be shared as a **submission**: it is checked and read by a person before it joins the shelf, the way every toy in Splashery is today. Equations stay the quick way to share; recipes are the deep one.

Until then, everything in Chapter 2 is in the source code, which is open under the MIT license: github.com/ryanjosephkamp/splashery.

Glossary

The main terms a newcomer meets in Chapters 1 to 10, in the order the chapters bring them up. On the other pages, a word with a dotted underline opens its entry.

Gaussian (peak-one bell)

A smooth weight that falls off exponentially with the squared distance from a center, measured in units of the widths. In this manual it peaks at 1.

Weight field, positive widths

A weight field is a number at every point in space: 1 at the center, falling off with distance. The three widths are a splat's sizes along its own axes; they must be above zero.

Mean, center μ

The position where the bell has its largest weight.

Standard deviation, axis width s

A length that measures distance along one of a splat's own axes. It is a yardstick, not a hard radius: the ideal weight is still about 0.14 at two widths.

Covariance Σ

A symmetric 3-by-3 matrix that holds the squared widths and the way the axes are turned, $\Sigma = R\,S\,S^{\mathrm{T}}\,R^{\mathrm{T}}$. Terms off the diagonal mean the axes are tilted.

Quadratic form

An expression in which every term has two factors of the input. In two dimensions, $ad_1^2 + 2bd_1d_2 + cd_2^2$. With 4 and 1 on the diagonal it is $4d_1^2 + d_2^2$, and setting it to 1 draws an ellipse. The exponent of a Gaussian holds one [5].

Matrix inverse

The matrix that undoes another one: multiplying a matrix by its inverse gives the identity matrix (1s on the diagonal, 0s everywhere else). A Gaussian bell uses the inverse Σ^{-1} of its covariance. Only a matrix whose determinant is not 0 has one.

Determinant

One number that sums up a square matrix. For a 2-by-2 covariance with entries a, b, b, c it is $ac - b^2$, which is also the product of its two eigenvalues. A determinant of 0 means the matrix is flat in one direction and has no inverse.

Eigenvalue, eigenvector

An eigenvector of a matrix is a direction the matrix only stretches, never turns; its eigenvalue says by how much. For a covariance, the eigenvectors are the axes of the bell's ellipse, and the eigenvalues are the squares of its widths along them.

Positive definite

A symmetric matrix whose eigenvalues are all above 0 (for a 2-by-2 matrix: a above 0 and the determinant above 0). A covariance has to be positive definite, so that the bell falls off in every direction instead of growing in one.

Tilt, rotation

Tilt is the off-diagonal terms of Σ: zero when a splat's axes line up with x, y, and z. Rotation is a splat's own turn. "Tilt lock" is a different thing: the camera control in Chapter 5.

Anisotropic, isotropic

Different widths along the axes, or equal widths in every direction. A plain kit splat is isotropic; the flat discs that cover a surface are anisotropic.

Radian

A way to measure an angle by the length of arc it cuts off on a circle of radius 1. A full turn is 2π (about 6.28) radians, a half turn is π, and a quarter turn is $\pi/2$. The equation language's `sin` and `cos`, and the angles in a recipe, use radians, not degrees.

Quaternion q

Four numbers that describe a rotation. They only describe a rotation when their length is 1, so a reader scales them to length 1 first.

Affine

A straight-line map plus a shift, $p \mapsto Ap + b$. Straight lines stay straight [6].

Footprint

The patch of the screen where one splat can add any color.

Jacobian J

The matrix of the projection's slopes at one point, used as a straight-line stand-in for the projection near the splat's center.

Opacity α, transmittance T

Opacity says how much a splat covers. Transmittance is how much of what lies behind still shows after the splats in front: the product of $(1 - a)$ over them.

Alpha compositing

Laying colors over one another in order, each weighted by its opacity and by how much of the pixel is still uncovered.

Anti-aliasing

Softening the jagged stair-steps on an edge by blending the pixels it only partly covers. Splashery's own option (off unless a labs switch turns it on) lowers each splat's opacity to make up for the energy that a little extra blur spreads out.

Spherical harmonics

A compact way to store how a color changes with the direction you look from. Many trained captures store them; a kit splat has one base color.

Recipe, capture

A recipe is a program that makes splats. A capture is a file of splats fitted to photographs of a real thing.

Kit, kit toy

The kit is Splashery's toy-building library. A recipe names shapes, colors, and moving parts, and the kit fills them with splats in the browser, spreading the budget evenly by area. A kit toy is built this way; a scan or capture is loaded from a file.

Smoothstep

The curve $3g^2 - 2g^3$, an S-shaped ramp from 0 to 1 that is flat at both ends, used to start and stop a motion gently [7].

Grammar, atom

A grammar is a list of rules for what counts as a valid program. In it, `=` is "is made of", `|` is "or", `{ }` repeats, `[ ]` is optional, and quotes mark text typed exactly. An atom is the smallest piece of an expression that can stand alone [8].

u, v, t, morph

u and *v* are the coordinates of the flat sheet of points that each splat gets a place on, before the equations bend it into shape. They are parameters of a parametric surface, not a substitution [9]. *t* is time: 0 to 2π in the equation language, and the toy's running time in a recipe's `drive`. A morph is a splat sliding from its own place to a target place.

Parametric surface

A surface drawn by giving *x*, *y*, and *z* each as a formula of two numbers, *u* and *v*. Each splat gets its own pair (*u*, *v*), and the equations carry it to a place in space. A sphere, a torus, and a Möbius strip are all parametric surfaces.

Kernel

The rule for how a splat's weight falls off across its patch. The site normally draws a cut-off Gaussian; a labs setting can use a sharper one. Neither is the ideal bell.

Part, token

A **part** is a group of a toy's splats that moves as one solid piece, such as a lid or a sail. Every splat carries the number of its part; number 0 is the body, so a toy has up to 15 more (Chapter 2). A **token** is one of up to 48 separate pieces, such as a game piece, that a recipe moves by its own offset.

Lever

A piece that a recipe can move or turn by its own offset, such as a piano key. A kit toy can have up to 96 levers in 6 groups, besides its parts and tokens.

Drive

The function in a recipe that runs every frame and says how the toy looks now: the angle of each part, the glow, the page of a sheet. It does not move splats itself; it hands numbers to the graphics chip.

Splat budget, tier

The number of splats the toy is asked to draw. It depends on the device: a tier (low, mid, high, or max) sets a default count and a ceiling, and a recipe may ask for more or fewer per base budget with its `density`. The counts run from 60,000 up to 400,000.

Pack

A file under `src/packs/` that holds the recipes of a group of toys.

Labs switch

The setting that shows the toys and tools not yet ready for everyone. Add `?labs=1` to the site's address to turn it on (the browser remembers it) and `?labs=0` to turn it off. A link to a labs toy opens it either way; the switch only decides what the shelf shows.

Seed

A number that starts a repeatable stream of pseudo-random numbers. The same seed gives the same toy every time, which is why a link can rebuild it.

Sorting, back to front

Splats are partly see-through, so they must be drawn from the farthest to the nearest for the blending to come out right. The engine sorts them for the camera; a toy that moves its splats may need them sorted again where they stand.

Uniform, shader

A **shader** is a small program that runs on the graphics chip for every splat. A **uniform** is a value the page hands it once a frame, the same for every splat, such as a part's rotation. Hands-on play, a recipe's drive, and the controls all reach the screen as uniforms.

Level of detail

Drawing less where little shows: a big structure gets fewer splats per atom, and a page far away gets a coarser sheet.

Web worker

A background thread of a page. Heavy jobs (reading a file, turning a page into splats, training) run in one so the page stays smooth.

Sheet, leaf, spine

A **sheet** is a flat place in a toy where a picture shows (Chapter 5). A **leaf** is a sheet that turns like a page, about a **spine**.

Hands-on, rigid body

Hands-on is the ✋ switch: a drag picks up the toy and a let-go tosses it. A **rigid body** is a thing that never changes shape, described by its place and its turn; most toys play as one (Chapter 6).

Position-based dynamics (XPBD)

A way to simulate physics by moving things to where they would be if nothing overlapped and every joint held, then working out their speeds from how far they moved. Splashery's own engine uses an extended form of it [11, 12].

Substep

One of the smaller steps a physics engine takes inside a single time step, to be more accurate. Hands-on steps 1/60 of a second at a time, in 8 substeps for a whole toy and 10 for a toy with pieces; smaller steps keep fast pieces from sinking into one another.

Joint: hinge, slider, dial, socket, break

The ways a part may move in hands-on play. A **hinge** swings about an axis, a **slider** moves along a line, a **dial** turns round and round, a **socket** lets a loose piece click back into its place, and a **break** lets a piece snap off when pulled hard. Each of the first three is moved by one number, an angle or a distance.

Node, skin

A **node** is one point of a rope or a cloth, held to its neighbors by links. A splat that follows two nodes (or four, for cloth) is called a **skin** splat: the rope bends and the splat stays whole.

Restitution (bounce)

The share of its speed a thing keeps after a bounce: a basketball dropped on a hard floor keeps about 0.82 to 0.88.

Toy radius

The unit of length hands-on play uses: the toy's own radius. Gravity, throws, and the play area are given in toy radii, so every toy falls the same way whatever its size on screen.

Ångström, B-factor

An **ångström** is a tenth of a nanometer; atoms in a bond are about 1 to 2 of them apart. A **B-factor** says how much an atom's place is blurred in a crystal, by motion or disorder.

CT, MRI

Two kinds of medical scan that measure the inside of a body slice by slice: CT (computed tomography) uses X-rays, and MRI (magnetic resonance imaging) uses magnetic fields and radio waves. Each slice is a grid of numbers, and a stack of slices is a volume of voxels.

Voxel

A volume pixel: one little cube of a 3D scan, with one number for how dense the material is there.

Window and level, Hounsfield unit

A **window** picks the range of values that show, from a lower to an upper level, and the rest is hidden or faded. In a CT scan the values are in **Hounsfield units**, on which water is 0 and air is about −1000.

Spectrogram

A picture of a sound over time: time runs along one side, pitch along the other, and brightness shows how loud each pitch is at that moment. The Song landscape toy turns the spectrogram of a song into a 3D field.

Point cloud, lidar, LAS and LAZ

A **point cloud** is a list of points with places and sometimes colors. **Lidar** makes one by timing laser pulses. **LAS** is the standard file for them; **LAZ** is the same file compressed.

Magnitude

A scale on which each step is a fixed ratio. An earthquake's magnitude goes up by one for about 32 times the energy. A star's magnitude goes the other way: the smaller (or more negative) the number, the brighter the star, and 6 is about the faintest the unaided eye sees under a dark sky.

Precession, sidereal time, J2000

Precession is the slow wobble of the Earth's axis, one turn in about 26,000 years, which shifts where the stars are. **Sidereal time** is time measured by the stars instead of the Sun, and tells which part of the sky is overhead. **J2000** is the moment (January 1, 2000, noon) a star catalog's positions are given for.

Snapshot, feed

A **feed** is a web address that serves current data, such as the earthquakes of the past week. A **snapshot** is a dated copy of that data that ships with the site, for when the feed cannot be reached.

Depth map, relative depth

A picture the same size as a photo whose every pixel holds how far away that part is. A depth model's map is **relative**: it says which parts are nearer than which, not how many meters. Often it is stored as *disparity*, which is large for near things.

Sharp view, relief

A second view of the photo toys that lays the photo, at its full resolution, on a **relief**: a surface lifted by the same depth the splats use. It is a textured surface, not splats.

Training, gradient descent

Gradient descent nudges numbers again and again in the direction that lowers an error. **Training** a splat scene is gradient descent on the splats' places, sizes, colors, and opacities until pictures drawn from them match the photos [1].

Structure from motion

Working out where the camera was for each of many overlapping photos, and where the points they show lie in space.

WebGPU, WebAssembly

WebGPU is the browser's newer way to use the graphics chip for general computation, which Video to 3D needs for training. **WebAssembly** is a compact code format that runs at near native speed; the depth model runs on it through ONNX Runtime Web.

PLY, SOG, SPZ, .splat

Splat file formats. **PLY** is the plain one that every tool reads and that keeps everything. **SPZ** is about a tenth of its size and keeps view-dependent color. **SOG** is the smallest and can keep view-dependent color (Splashery's Splat toolkit writes it with base colors only). **.splat** is simple and keeps base colors only.

Module, finder pattern, quiet zone

A **module** is one cell of a QR code. The three big squares in the corners are the **finder patterns**, which let a reader find and orient the code. The **quiet zone** is the blank border, four modules wide.

Error correction, Reed–Solomon, codeword

A QR code holds extra **codewords** (8 modules each) made with a **Reed–Solomon** code [19], so a reader can rebuild the text if some are lost. The level (L, M, Q, or H) sets how much damage it can repair: about 7, 15, 25, or 30 percent of the codewords lost.

QR version

The size of a QR code, from 1 (21 modules on a side) to 40 (177). A longer text needs a higher version.

Halftone, error diffusion

Showing a picture with only dark and light dots. **Error diffusion** decides each dot by passing the error of the last choice on to the pixels beside it.

Check digit

A digit of a barcode worked out from the others, so a reader can tell when it has misread one.

Fragment, deflate, base64url

The **fragment** of a link is everything after the `#`; a browser never sends it to a server. **Deflate** is the compression of zip files and PNG pictures; a Splashery link uses its raw form. **Base64url** spells bytes with the 64 letters, digits, `-`, and `_` that are safe in an address.

Scene

Everything needed to rebuild a view: the toy, its options, the look, the camera, and any paint, written as JSON. A link is a scene; a scene file is the same scene saved as a file.

Further reading

Each source below was opened and checked on its own page (the first five on October 4, 2026, the rest on October 11, 2026). Authors and titles are as the source's page gives them. A venue is named only where the page names one; the others are preprints on arXiv. These are links only: nothing here is copied.

Where splats come from

- Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. **3D Gaussian Splatting for Real-Time Radiance Field Rendering.** SIGGRAPH 2023 (ACM Transactions on Graphics). Project page · Paper. The starting point.
- Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. **EWA Volume Splatting.** Proceedings of IEEE Visualization 2001. Paper. The footprint step in Chapter 1.
- Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. **Mip-Splatting: Alias-free 3D Gaussian Splatting.** CVPR 2024 (Oral, Best Student Paper). Project page.
- Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. **2D Gaussian Splatting for Geometrically Accurate Radiance Fields.** SIGGRAPH 2024. Project page. A different method from Splashery's flat one-photo fit.
- Antoine Guédon and Vincent Lepetit. **SuGaR: Surface-Aligned Gaussian Splatting for Efficient 3D Mesh Reconstruction and High-Quality Mesh Rendering.** CVPR 2024. Project page.

Robotics and other uses

- Nikhil Keetha, Jay Karhade, Krishna Murthy Jatavallabhula, Gengshan Yang, Sebastian Scherer, Deva Ramanan, and Jonathon Luiten. **SplaTAM: Splat, Track & Map 3D Gaussians for Dense RGB-D SLAM.** CVPR 2024. Paper. A robot camera maps a room with splats as it moves.
- Hidenobu Matsuki, Riku Murai, Paul H. J. Kelly, and Andrew J. Davison. **Gaussian Splatting SLAM.** CVPR 2024 (Highlight). Paper. The same problem with one ordinary camera.
- Timothy Chen, Ola Shorinwa, Joseph Bruno, Aiden Swann, Javier Yu, Weijia Zeng, Keiko Nagami, Philip Dames, and Mac Schwager. **Splat-Nav: Safe Real-Time Robot Navigation in Gaussian Splatting Maps.** arXiv preprint, 2024. Paper. A robot plans a safe path through a splat map.

- Yuhang Zheng, Xiangyu Chen, Yupeng Zheng, Songen Gu, Runyi Yang, Bu Jin, Pengfei Li, Chengliang Zhong, Zengmao Wang, Lina Liu, Chao Yang, Dawei Wang, Zhen Chen, Xiaoxiao Long, and Meiqing Wang. **GaussianGrasper: 3D Language Gaussian Splatting for Open-vocabulary Robotic Grasping.** arXiv preprint, 2024. Paper. A robot is told in words what to pick up.
- Mazeyu Ji, Ri-Zhao Qiu, Xueyan Zou, and Xiaolong Wang. **GraspSplats: Efficient Manipulation with 3D Feature Splatting.** arXiv preprint, 2024. Paper.
- Yunzhi Yan, Haotong Lin, Chenxu Zhou, Weijie Wang, Haiyang Sun, Kun Zhan, Xianpeng Lang, Xiaowei Zhou, and Sida Peng. **Street Gaussians: Modeling Dynamic Urban Scenes with Gaussian Splatting.** arXiv preprint, 2024. Paper. Driving scenes, with moving cars.
- Yifan Liu, Chenxin Li, Chen Yang, and Yixuan Yuan. **EndoGaussian: Real-time Gaussian Splatting for Dynamic Endoscopic Scene Reconstruction.** arXiv preprint, 2024. Paper. Surgery video, in medicine.

Moving, talking, and generated splats

- Tianyi Xie, Zeshun Zong, Yuxing Qiu, Xuan Li, Yutao Feng, Yin Yang, and Chenfanfu Jiang. **PhysGaussian: Physics-Integrated 3D Gaussians for Generative Dynamics.** CVPR 2024. Paper. Splats that move by physics, as in Chapter 6's spirit.
- Jonathon Luiten, Georgios Kopanas, Bastian Leibe, and Deva Ramanan. **Dynamic 3D Gaussians: Tracking by Persistent Dynamic View Synthesis.** arXiv preprint, 2023. Paper.
- Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. **4D Gaussian Splatting for Real-Time Dynamic Scene Rendering.** CVPR 2024. Paper.
- Shenhan Qian, Tobias Kirschstein, Liam Schoneveld, Davide Davoli, Simon Giebenhain, and Matthias Nießner. **GaussianAvatars: Photorealistic Head Avatars with Rigged 3D Gaussians.** arXiv preprint, 2023. Paper. Heads that can be animated.
- Minghan Qin, Wanhua Li, Jiawei Zhou, Haoqian Wang, and Hanspeter Pfister. **LangSplat: 3D Language Gaussian Splatting.** CVPR 2024. Paper. Searching a splat scene with words.
- Jiaxiang Tang, Jiawei Ren, Hang Zhou, Ziwei Liu, and Gang Zeng. **DreamGaussian: Generative Gaussian Splatting for Efficient 3D Content Creation.** arXiv preprint, 2023 (camera-ready 2024). Paper. Making a splat from a prompt or a picture.

- Tong Wu, Yu-Jie Yuan, Ling-Xiao Zhang, Jie Yang, Yan-Pei Cao, Ling-Qi Yan, and Lin Gao. **Recent Advances in 3D Gaussian Splatting.** arXiv preprint, 2024. Paper. A survey of the field.

Open projects and tools

- **gsplat**, an open-source library for training and rendering splats on the graphics chip. github.com/nerfstudio-project/gsplat. Vickie Ye and colleagues also describe it in a paper, "gsplat: An Open-Source Library for Gaussian Splatting" (arXiv preprint, 2024): arxiv.org/abs/2409.06765.
- **Brush**, a 3D reconstruction engine that uses Gaussian splatting and runs on desktop, Android, and in browsers (Apache-2.0). github.com/ArthurBrussee/brush.
- **SuperSplat**, a free, open-source tool for inspecting, editing, optimizing, and publishing splats (MIT). github.com/playcanvas/supersplat.
- **Spark**, a splat renderer for THREE.js (MIT). github.com/sparkjsdev/spark.
- **antimatter15/splat**, a small WebGL viewer that renders splat scenes in the browser without dependencies (MIT). github.com/antimatter15/splat.
- **KHR_gaussian_splatting**, a glTF extension for storing Gaussian splats, which its page lists as ratified by the Khronos Group. github.com/KhronosGroup/glTF (extensions/2.0/Khronos/KHR_gaussian_splatting).

References

Numbered sources, cited in the text by their number in square brackets. Each entry gives the authors, the title, where it appeared, the year, and a link. "Further reading" above lists more papers on splats.

1. Kerbl, B., Kopanas, G., Leimkühler, T., and Drettakis, G. "3D Gaussian Splatting for Real-Time Radiance Field Rendering." *ACM Transactions on Graphics* (SIGGRAPH), 2023. arxiv.org/pdf/2308.04079
2. Zwicker, M., Pfister, H., van Baar, J., and Gross, M. "EWA Volume Splatting." *Proceedings of IEEE Visualization*, 2001. cs.umd.edu/~zwicker/publications/EWAVolumeSplatting-VIS01.pdf
3. Porter, T., and Duff, T. "Compositing Digital Images." *Computer Graphics* (SIGGRAPH) 18(3), 1984, pp. 253–259. doi.org/10.1145/964965.808606 (some networks refuse that address; the author-hosted copy is keithp.com/~keithp/porterduff/p253-porter.pdf)
4. "Multivariate normal distribution." *Wikipedia*, the free encyclopedia. en.wikipedia.org/wiki/Multivariate_normal_distribution
5. "Quadratic form." *Wikipedia*, the free encyclopedia. en.wikipedia.org/wiki/Quadratic_form
6. "Affine transformation." *Wikipedia*, the free encyclopedia. en.wikipedia.org/wiki/Affine_transformation
7. "Smoothstep." *Wikipedia*, the free encyclopedia. en.wikipedia.org/wiki/Smoothstep
8. "Extended Backus–Naur form." *Wikipedia*, the free encyclopedia. en.wikipedia.org/wiki/Extended_Backus–Naur_form
9. "Parametric surface." *Wikipedia*, the free encyclopedia. en.wikipedia.org/wiki/Parametric_surface
10. Kamp, R. J. "Writing toys for Splashery packs" (`docs/PACKS.md`). *Splashery* source code, 2026. github.com/ryanjosephkamp/splashery/blob/main/docs/PACKS.md
11. Müller, M., Macklin, M., Chentanez, N., Jeschke, S., and Kim, T.-Y. "Detailed Rigid Body Simulation with Extended Position Based Dynamics." *Computer Graphics Forum*, 2020. doi.org/10.1111/cgf.14105
12. Macklin, M., Müller, M., and Chentanez, N. "XPBD: Position-Based Simulation of Compliant Constrained Dynamics." *Proceedings of the 9th International Conference on Motion in Games (MIG '16)*, ACM, 2016, pp. 49–54. matthias-research.github.io/pages/publications/XPBD.pdf

13. Yang, L., Kang, B., Huang, Z., Zhao, Z., Xu, X., Feng, J., and Zhao, H. "Depth Anything V2." *Advances in Neural Information Processing Systems* (NeurIPS), 2024. arxiv.org/abs/2406.09414

14. Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N., and Bourne, P. E. "The Protein Data Bank." *Nucleic Acids Research* 28(1), 2000, pp. 235–242. doi.org/10.1093/nar/28.1.235

15. U.S. Geological Survey. "GeoJSON Summary Format." Earthquake Hazards Program. earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php

16. "HYG Database." The Astronomy Nexus (astronexus.com), CC BY-SA 4.0. www.astronexus.com/projects/hyg

17. Meeus, J. *Astronomical Algorithms*, 2nd ed. Willmann-Bell, 1998. (A book; the link is its catalog record.) openlibrary.org/isbn/9780943396613

18. Jet Propulsion Laboratory, Solar System Dynamics group. "Approximate Positions of the Planets." ssd.jpl.nasa.gov/planets/approx_pos.html

19. Reed, I. S., and Solomon, G. "Polynomial Codes over Certain Finite Fields." *Journal of the Society for Industrial and Applied Mathematics* 8(2), 1960, pp. 300–304. doi.org/10.1137/0108018

20. Project Nayuki. "QR Code generator library." MIT License. www.nayuki.io/page/qr-code-generator-library

21. Splat.js, an MIT-licensed library for training Gaussian splats in the browser (Stratum1 GmbH). github.com/arrival-space/splat.js

22. Mozilla. "PDF.js." Apache License 2.0. mozilla.github.io/pdf.js

Quick reference

One splat

$$G(p) = \exp(-\tfrac{1}{2}\,(p-\mu)^{\mathrm{T}}\,\Sigma^{-1}\,(p-\mu)), \quad \Sigma = R\,S\,S^{\mathrm{T}}\,R^{\mathrm{T}}$$

$$\Sigma' = J\,W\,\Sigma\,W^{\mathrm{T}}\,J^{\mathrm{T}}, \quad C \leftarrow a\,c + (1-a)\,C \text{ (back to front)}$$

A splat program

```
x = …        y = …        z = …        # in u, v and t (y is up)
u = 0 .. 2pi                           # ranges: numbers only
v = 0 .. pi                            # default 0 .. 1
hue = …                                # 0 red, 1/3 green, 2/3 blue
r = …   g = …   b = …                  # or colors from 0 to 1
size = 0.05                            # about the bell's σ
count = 4000                           # 100 to 10,000
spread = grid                          # or random
```

Functions: sin, cos, tan, asin (arcsin), acos (arccos), atan (arctan), sinh, cosh, tanh, exp, log (ln), log10, sqrt, abs, floor, min, max. **Constants**: pi, e. **Time**: *t* runs from 0 to 2π per tap. At most 240 characters an expression (each end of a range counts on its own).

A picture toy

```
pictures: { sample: () => "assets/…", accept: ["pdf", "image", "gif", "video"] },
input: { title, media: { accept } },     // or placeholder, button, read, shown; or pad
tiltLock: true,                          // a drag turns it only left and right
build(k) { k.sheet({ id, center, width, height, leaf }); k.spine({ at, axis, dir }); },
drive(t, c, out, info) {
  out.sheets[id] = { page, visible };    // pages count from 0
  out.leaves[slot] = { angle, curl };    // radians; 0 flat, pi turned over
  info.data.pictures                     // page, count, go(n), next(), prev(), togglePlay()
}
```

A recipe (a toy)

```
export const RECIPES = { "my-toy": {
  options: [{ key, label, type, default }],   // saved in the link
  controls: [{ key, label, type: "slider" | "toggle" | "pulse", ease }],
  action: { key, label },                     // the tap
  build(k, o) { /* k.sphere, k.box, k.part(name, { pivot, axis }), k.add(shape, { part, pos, color }) */
```

```
},
  drive(t, c, out, info) { out.parts[name] = { angle }; },  // every frame
  hands: { pieces, joints, ropes, material },  // the ✋ switch (Chapter 6)
  handsOn: false, tiltLock: true, turntable: false,
}};
```

The controls

Drag, pinch, scroll	Turn, zoom (two fingers roll only after a twist of about 17°)
Shift or Option/Alt and a drag; two-finger drag	Move the toy across the screen
R, ↺, double-tap	Reset the view
1 to **5**	Orbit, Poke, Paint, Magnet, Clay
F, **Esc**	Focus mode on, and back
[, **G**	Fold the side panel, open the gallery (on a computer)
✋	Hands-on: drag to pick up, let go to toss, ↺ to send home
Padlock, record player	Tilt lock (a drag turns left and right only), turntable when idle
`?labs=1` or `?labs=0`	Show or hide the toys not yet ready for everyone

Key numbers

What	Number	Chapter
Splats a device is asked for	60,000 up to 400,000, by tier	2
Splats of an equation toy	100 to 10,000	3
Characters in one equation field	240	3

What	Number	Chapter
Option keys of the equation toy	16	3
Moments of *t*	12	3
Parts, besides the body	15	2
Tokens, levers, lever groups	48, 96, 6	2
Hands-on: gravity, throw cap	26 and 4 toy radii per second (squared for gravity); 4 is the base cap, and a material can scale a throw to 5	6
Hands-on: collision points of a toy	64, from up to 6,000 splats	6
Stars in the night sky	5,071	7
Largest molecule, volume	250,000 atoms; 8 million voxels	7
Depth model	27 MB, loaded only for your own photo	8
Moving photo, video clip length	8 seconds for a GIF or a short clip (a longer video plays whole)	8
QR code error correction	L 7%, M 15%, Q 25%, H 30% of codewords	9
A link	Up to 12,288 characters of scene; version 21 QR code up to 711 characters at M	9

What goes where

Splat files (`.ply` , `.sog` , `.splat` , `.spz`), **models** (glTF, GLB, OBJ, STL), **pictures** (PDF, photo, GIF, video), **data** (CSV, DICOM, NIfTI, TIFF, LAS, LAZ, PDB) are opened on your device and never uploaded. A link carries settings, not files.

The Tinkerer's Manual, first published September 28, 2026 and updated October 11, 2026. Part of Splashery, a toy box of 3D Gaussian splats that runs in your browser. Every picture here was made with Splashery itself.