# typstage

Version 0.1.0

Animated HTML presentations from a single Typst file, and a PDF handout from the same source. Typst typesets, the browser moves: magic-move morphing, step-by-step reveals, slide transitions, media and a speaker view in the same file.

`typstage` typesets presentations with Typst and moves them in the browser. Every slide is set as SVG, so the arrangement is the one the PDF has, and whatever is meant to move is registered for it: reveals, magic move, slide transitions, media. One source yields an animated HTML talk, a slide deck as PDF, and a handout to write on.

AUTOREN Alexander Schulz <@Loewe1000>

VERWEISE GitHub · Examples · Deutsch

# Inhalt

# 1 What this package is

`typstage` turns a single Typst file into an animated presentation for the browser, and a PDF from the same source. **Typst typesets, the browser moves.** Every slide is set by Typst as SVG, so the arrangement in the browser is the one on paper. Whatever is meant to move is marked in the source, and a small runtime animates it.

A slide therefore stays a slide, not a stack of intermediate states. The PDF has one page per slide, not one per step, and whatever belongs to the motion alone falls away on paper.

## 1.1 Five words this manual uses

**Slide** — One picture, typeset once by Typst. One page of the PDF, one `.ts-slide` in the HTML.

**Step** — One press of the arrow key. A slide can hold several; at the last one the next press moves on. The address bar counts steps, the footer counts slides.

**Element** — A piece of a slide that the runtime may touch. `anim`, `stagger`, `alternatives`, `morph`, `scene`, `embed`, `video` and `flipbook` all produce one.

**Morph** — The same named element twice – on two adjacent slides, or on two steps of one slide. Between them it flies, glyph by glyph where it can.

**Speaker view** — The same file opened a second time with `#speaker` on the address. It carries the note, the clock and the next step, and it draws on the slide the room sees.

## 1.2 Where it sits among the others

`touying` and `polylux` are mature, have far more themes, and make PDF. For a normal PDF talk, take one of those. `reveal.js`, `Slidev` and `Quarto` animate in the browser, but their layout is HTML's, not Typst's.

Nearest are the Typst packages that write HTML themselves. `touying-exporter` renders one SVG per slide and packages the sequence with `impress.js`. `slipst` follows slipshow and gives up the fixed-size slide altogether: there „slips“ scroll from top to bottom. On the PDF side, `mosaic` is worth a look – it cuts its slides from the same headings this package does, `=` for the section and `==` for the slide – and so is `slydekit`. Three of the seventeen example decks are adaptations of mosaic decks, so that part of the comparison is on the screen rather than in my prose.

What this package does instead: a named piece stands in one place on slide n and elsewhere on slide n+1, and it flies between the two – glyph by glyph, so an equation visibly rewrites itself. Weaker forms cover the rest: a page that turns, a cross-fade, whole slides pushed around by a script. What carries all of it is one SVG per *state* rather than per slide, so between two states there is something left that can fly.

> **ACHTUNG**
> The price, stated before the first line of code: the slides are SVG outlines. Nothing in the browser is selectable or searchable, and a screen reader sees nothing at all. For some talks that is too high; there is a chapter on it further down.

This manual is ordered by intent rather than by function:

1. **Your first presentation** — from the empty file to a running HTML
2. **One deck, from start to finish** — one talk, built to the end
3. **Revealing a slide step by step** — `pause`, `stagger`, `anim`, `alternatives`
4. **Showing instead of claiming** — an applet, a video, a flip book
5. **GeoGebra** — constructions that follow the steps of the slide
6. **Desmos** — the same road, a different calculator

7. **Developing a calculation** — magic move across several slides
8. **Giving the talk** — keys, touch, the overview, the speaker view
9. **Three outputs from one source** — talk, slide deck, handout
10. **Making it your own** — themes, colours, canvas, building blocks
11. **Handing it on** — one file, assets, hosting
12. **What it cannot do** — reach, accessibility, size
13. **When nothing happens** — the traps, in the order they are usually hit
14. **API reference** — every function, from the source

> **HINWEIS**
> The typeset examples here are paper and show the final state, everything at once. What happens one after another in the browser is said in the text beside them.
>
> Every `typ` listing is compiled against the real package before publishing. That catches a listing which no longer compiles – not one that compiles and does the wrong thing.

# 2 Your first presentation

A complete, presentable talk in ten minutes.

## 2.1 One file is enough

An import, a show rule, headings. This file is complete and can be typed out as it stands:

```
#import "@preview/typstage:0.1.0": *

#show: presentation.with(
  title: [The Pythagorean Theorem],
  subtitle: [A derivation in four steps],
  author: [Mathematics · Year 9],
  date: datetime.today(),
  transition: "slide",
)

= What this is about

== The claim

#speaker-note[Show the dissection first, then the formula, not the other way round.]

In a right-angled triangle the two shorter sides together carry as much area
as the longest one.

#pause

And that is the formula: $a^2 + b^2 = c^2$
```

A first-level heading is a section slide, a second-level heading is a slide, and the text below it is its body. That is the whole structure.

## 2.2 More than two levels

By default `=` becomes a section slide and `==` a slide. `slide-level` moves that cut: a heading **above** it becomes a section slide, a heading at it or below it becomes a slide.

```
#show: presentation.with(title: [Analysis I], slide-level: 3)
= Part I -- Limits
== Sequences
=== What a sequence is
A map from the naturals into the reals.
=== Convergence
For every epsilon there is an N.
== Series
=== Partial sums
The sum of the first n terms.
```

`= Part I` and `== Sequences` each become a section slide, every `===` becomes a slide. Every section heading is its own transition slide, so there is nothing to switch on.

`slide-level: 1` makes every heading a slide; the deck then has no structure level at all.

The five bundled themes draw a deeper level more quietly: the title gets smaller, and above it stands what the section hangs under. What the deck knows about its structure is in `info()` – see „`info()`: what the deck knows about itself“.

### 2.2.1 Text that belongs to no slide

A section slide is a whole picture the theme draws; it has no body. Text between a section heading and the next heading therefore belongs to no slide, and stops the compile instead of silently disappearing.

A sentence between `= The proof` and `== The dissection` aborts with `content between the heading "The proof" and the next one belongs to no slide`:

```
#show: presentation.with()
= The proof
This sentence belongs to no slide and stops the compile.
== The dissection
```

It belongs under the slide heading:

```
#show: presentation.with()
= The proof
== The dissection
This sentence belongs to the slide and gets typeset.
```

Text **before** the first heading is refused the same way, as long as the deck has at least one heading. Both rules apply to heading notation only.

## 2.3 When the slides are computed

Headings created while the document is set become slides too. A loop over a list gives one slide per entry:

```
#for element in ("Water", "Air", "Earth") [
  == #element
  Something about #element.
]
```

Where the slides come entirely from data, hand them over one by one instead. Each slide is a function call, and a list of slides spreads with `..` like any other array:

```
#presentation(
  title-slide(title: [The Pythagorean Theorem], author: [A. Schulz]),
  section[The proof],
  slide([The dissection], note: [Show the square first.])[
    The text of the slide.
  ],
)
```

Both spellings give the same output; `presentation` tells them apart from its arguments. The heading form is the ordinary case.

## 2.4 Two compilations

The same file gives two outputs. The flags decide which:

```
typst compile talk.typ talk.html --format html --features html
typst compile talk.typ talk.pdf
```

> **ACHTUNG**
> Without `--features html`, HTML export is unavailable, and Typst's error message reads like a mistake in your file rather than a missing flag. The feature is experimental in Typst itself, not in this package.

## 2.5 Looking at it

The HTML is one file. Double-click it and it runs: no server, no network, nothing loaded afterwards.

Arrow keys page. `?` shows every key, `o` opens the overview, `f` goes full screen, and `n` opens the speaker view in a second window.

# 3 One deck, from start to finish

One talk in a single file, from the empty line to the handout. Every step adds exactly one thing, and together they cover what an ordinary deck needs.

The subject is a school exercise: **How tall is the tower?** A pole 1.20 m high casts a shadow of 0.90 m; the tower casts a shadow of 21 m. Find its height.

## 3.1 The empty file

Two lines are a deck: one fetches the package, one says this document is a presentation.

```
#import "@preview/typstage:0.1.0": *
#show: presentation.with(title: [How tall is the tower?])
```

It is compiled twice, from the same file:

```
typst compile tower.typ tower.html --format html --features html
typst compile tower.typ tower.pdf
```

Open the HTML and page with the arrow keys. So far there is only the title slide – `title:` alone produces it.

## 3.2 The first slide

`==` is a slide, the text below it its body; `=` is a section slide.

```
= The question

== A pole and a tower

A pole 1.20 m high casts a shadow of 0.90 m.
The tower casts 21 m. How tall is it?
```

No more structure than this is needed.

## 3.3 What is to appear one after another

`stagger` splits a bullet list at its items: one step per item.

```
== What we see

#stagger[
  - The sun stands equally high for both.
  - So the angle is the same.
  - So the triangles are similar.
]
```

The slide now has four steps: the body and the three points.

## 3.4 A box that has to stick

The sentence that matters does not belong in the list. `callout` sets it apart, with a bar down its left side.

```
== What we see
```

```
#stagger[
  - The sun stands equally high for both.
  - So the angle is the same.
  - So the triangles are similar.
]

#callout[
  In similar triangles corresponding sides stand in the same ratio.
]
```

The caption of the box follows the document language. `title:` changes it, `title: none` leaves it off.

## 3.5 The formula that rewrites itself

The same formula stands on two slides, and between them it flies – glyph by glyph, as far as they recognise one another. Both slides call it by the same name; a label is all it takes.

```
== The ratio

#align(center, morph(<tower>, $ h / 21 = 1.2 / 0.9 $))

== Solved for h

#align(center, morph(<tower>, $ h = 21 dot 1.2 / 0.9 $))
```

In the browser `h`, the fraction bar and the numbers travel to their new place instead of vanishing and coming back. On paper the chain becomes the calculation, one slide per line.

## 3.6 A note only you see

`speaker-note` belongs to the slide and appears nowhere on the screen.

```
== The result

#statement[$ h = 28 "m" $]

#speaker-note[
  Let them work it out first, then show it. Anyone saying 28 has rounded --
  28.0 is more precise than the measurement allows.
]
```

It appears in the speaker view – opened in a second window with `n` – and in the handout, beside its slide.

## 3.7 The handout

One argument turns the deck into a sheet to write on: three slides per page, the note beside each one, ruled lines beside any slide that has none.

```
#import "@preview/typstage:0.1.0": *
#show: presentation.with(
  title: [How tall is the tower?],
  handout: 3,
)
```

## 3.8 The whole source

Nothing in it that was not explained above.

```
#import "@preview/typstage:0.1.0": *

#show: presentation.with(
  title: [How tall is the tower?],
  author: [Year 9],
  theme: themes.lesson,
)

= The question

== A pole and a tower

A pole 1.20 m high casts a shadow of 0.90 m.
The tower casts 21 m. How tall is it?

== What we see

#stagger[
  - The sun stands equally high for both.
  - So the angle is the same.
  - So the triangles are similar.
]

#callout[
  In similar triangles corresponding sides stand in the same ratio.
]

= The calculation

== The ratio

#align(center, morph(<tower>, $ h / 21 = 1.2 / 0.9 $))

== Solved for h

#align(center, morph(<tower>, $ h = 21 dot 1.2 / 0.9 $))

== The result

#statement[$ h = 28 "m" $]

#speaker-note[
  Let them work it out first, then show it. Anyone saying 28 has rounded --
  28.0 is more precise than the measurement allows.
]
```

**TIPP**

Two things worth a look next: `side-by-side` – a drawing on the left, the words on the right – and `theme:` , which changes the whole look without moving a line of content.

# 4 Revealing a slide step by step

A slide that unfolds in front of the room instead of standing there finished.

## 4.1 Which tool for what

Six building blocks cover nearly everything, and they mix on one slide.

| Tool | For what |
|---|---|
| `#pause` | The slide unfolds from top to bottom, with nothing wrapped around anything. The commonest case. |
| `stagger[…]` | A list point by point, bullet and text together. Also for several blocks in sequence. |
| `anim(…)` | One piece on one step, with a motion of its own. The tool wherever `#pause` cannot reach: grid cells, tables, boxes. |
| `alternatives(…)` | Several versions of the same thing in the same place, each replacing the one before. |
| `build(…)` | A drawing that comes into being in stages – one CeTZ line, one lilaq data series, one label after another. |
| `scene(…)` | A drawing that depends on a value. For everything that **moves** rather than being added. |

Beside them stand `tiles` for a grid that staggers itself, and `morph` for things that fly between two slides.

## 4.2 The step cursor

Every slide carries a step cursor. `at` defaults to `auto`, the next free step, so consecutive reveals number themselves and most slides hold no number at all.

```
== Three things
#anim[first]            // step 1
#anim[second]           // step 2
#anim(at: 4)[late]      // 4
#anim[after that]       // 5
```

The spellings of `at`:

| Written | Meaning |
|---|---|
| `auto` | the next free step (the default) |
| `3` | from step three on, the same as `"3-"` |
| `(2, 5)` | on step two and on step five |
| `"2-"` | from step two on |
| `"1-2"` | on steps one and two, not after that |
| `"2,4"` | on step two and on step four |
| `"-2"` | from the start until step two |
| `"3"` | exactly on step three |

A bare number is an open end: what is there once stays to the end of the slide. A closed spelling such as `"1-2"` or `"3"` lets the element disappear again, and then `exit` applies.

## 4.3 A slide without a single number

`#pause` needs no counting. It cuts the body where it stands, and everything after it arrives one step later:

```
== What we know

The two legs carry as much area as the hypotenuse.

#pause

$ a^2 + b^2 = c^2 $

#pause

And that is enough to compute the third side from two of them.
```

> **TIPP**
> `#pause` splits the body, so it works between blocks but not inside a grid cell or a table – there is nothing there to cut. `anim` goes anywhere content goes.

## 4.4 A list point by point

`stagger` takes a list and reveals it item by item, bullet and text together:

```
#stagger[
  - What the room already knows
  - What it is about to learn
  - What it will be able to do afterwards
]
```

`stride: 2` puts two items on each step, `stride: 0` all of them on one. `start` sets the first step, `enter` the motion, `stagger` the delay in milliseconds between neighbours, `spacing` the distance between items, `dim` lets each point step back once the next arrives.

> **TIPP**
> `stagger` also takes several blocks instead of one list. Each block is then one step – three paragraphs or three pictures in turn, without three `anim` calls.

`dim: true` turns the sequence into a walk: the point being discussed stands there, the ones before it stay legible but muted.

```
#stagger(dim: true)[
  - What the room already knows
  - What it is about to learn
  - What it will be able to do afterwards
]
```

Each point then holds its own step and rests in `after: "dimmed"` (see „The muted resting state“). Two consequences, both intended: the last point dims too once the slide has a step after it, and `stride: 0` dims them all together.

### 4.4.1 What stands beside a piece

`stagger-layer` hangs something on the step of one particular piece – the annotation beside a calculation, say. The stagger needs a name: `name:` says it, and a `morph:` written as a name says it too.

```
#stagger(morph: "rewrite",
  $ x^2 + 6x + 2 = 0 $,
  $ x^2 + 6x = -2 $,
  $ (x + 3)^2 = 7 $,
)

#stagger-layer("rewrite", 2)[$| -2$]
```

The layer stays from its piece to the end of the slide, as `cue-layer` and `scene-layer` do, and carries no morph name: what flies is the piece, the annotation merely appears beside it. The stagger has to stand **before** its layers, because a layer looks up which step its piece was given.

> **ACHTUNG**
> `spacing:` applies to the list branch only. Where the pieces stand on their own, ordinary block spacing decides.

## 4.5 Revealing in the order it is called out

Some points have no order. What a graph shows, what stands out in an experiment – a class names those as they come, and a deck that reveals them in **its** order makes the teacher wait or reshuffle. `cue` turns that round: the digits `1` to `9` reveal whatever was just named.

```
#cue("readings", start: 2)[
  - positive and negative values
  - lowest and highest value
  - falling and rising
]
```

The group takes a name so that `cue-layer` can point at it. It owns as many steps as it has points, whatever the order, so the progress bar, `info().step.total`, the overflow check and the handout are untouched. The list keeps its reading order: a point not yet named holds its place, so nothing jumps when it arrives.

### 4.5.1 What appears together with a point

`cue-layer` hangs something on the same step – a drawing layer, a picture, a sentence beside it:

```
#cue-layer("readings", 1, [and what goes with it])
```

The point and the layer share a step, so moving the step moves both, and you can hang as much on a point as you like. The group has to stand **before** its layers; otherwise the package says so.

> **TIPP**
> For a CeTZ drawing that grows with the points, draw every layer as its own complete drawing and hide the rest with `cetz.draw.hide(rest, bounds: true)`, so that it still counts towards the bounds. All layers then lie exactly on top of each other and the graph holds still, in whatever order it grows. A layer carries **only its own contribution**, no grid and no base curve, or the layer set last paints over the first. Where the drawing is to grow in written order, use `build` instead.

> **HINWEIS**
> The forward arrow reveals the next point **not yet named**, so paging alone behaves like a staggered list, and arrow and digits mix freely. Only once the group is full does the arrow carry on. One step back frees the point named last, and leaving the slide backwards leaves the group untouched for the next visit. In the speaker view every point still open stands there pale, with its digit on the bullet; in the hall it is invisible.

## 4.6 One piece on a step of its own

`anim` wraps exactly what should appear and says when:

```
#side-by-side(
  card(title: [Before])[The old way.],
  anim(at: 2, enter: "fade-left", card(title: [After])[The new one.]),
)
```

### 4.6.1 Entrance and exit

`enter` and `exit` name the motion. Twelve of them exist:

| Written | What happens |
|---|---|
| `"fade"` | opacity alone |
| `"fade-up"` | from a little below, the default for an entrance |
| `"fade-down"` | from a little above |
| `"fade-left"`, `"fade-right"` | from the side |
| `"scale"` | grows into place |
| `"scale-down"` | shrinks into place |
| `"blur"` | out of the blur |
| `"rise"` | from below and slightly smaller, the loudest of them |
| `"draw"` | it draws itself – see „A path that draws itself“ |
| `"none"` | it is simply there |
| `"hold"` | not an exit but a wait: the piece stays until the next one is there. As an `enter` it is the same as `"none"` |

`duration` is in milliseconds and `auto` takes the presentation's. `delay` holds the start back, which lets two elements on the same step arrive one after the other.

**A name the package does not know is an error at compile time**, as it is for `easing`: a typo would otherwise render as `"fade"` in silence.

```
#anim(enter: "fdae-up")[A typo.]   // error at compile time
```

### 4.6.2 The curve

Everything this package moves runs on one curve: slow off the mark, brisk through the middle, soft at the end. `easing` hands a different one to a single element.

```
#anim(result, enter: "rise", easing: "out-back")
#stagger(stride: 0, stagger: 60, easing: "out-quad")[
  - first this
  - then that
]
```

It stands wherever `duration` stands: on `anim`, `stagger`, `alternatives` and `build`, and it covers the entrance, the departure and the dimming. Not the slide transition, and not the flight of a magic move, which has two ends.

| Name | Curve |
|---|---|
| `"standard"` | this package's own curve, written out – the same as saying nothing |
| `"linear"` | even, with no run-up and no run-out |
| `"ease"`, `"ease-in"`, `"ease-out"`, `"ease-in-out"` | the four the Web Animations API knows by itself |
| `"in-quad"`, `"out-quad"`, `"in-out-quad"` | gentle |
| `"in-cubic"`, `"out-cubic"`, `"in-out-cubic"` | more pronounced |
| `"in-expo"`, `"out-expo"`, `"in-out-expo"` | sharp – nearly everything happens at one end |
| `"in-back"`, `"out-back"`, `"in-out-back"` | winds up and overshoots |

`in` means slow off the mark, `out` soft at the end; for an entrance `out` is nearly always right, because the eye watches the ending. **A name that does not exist is an error at compile time**, and the message lists the choices.

```
#anim(result, easing: "out-bounce")   // an error at compile time
```

**The three `back` curves go past their mark.** On a travel that is the swing back; on opacity the browser clips whatever reaches past 1, so `"out-back"` on a plain `"fade"` is merely a faster `"fade"`. Use it with an effect that travels: `"rise"`, `"scale"`, `"fade-up"`. Springs and bounces – `elastic`, `bounce` – do not exist here: they are not cubic Bézier curves, and the Web Animations API knows only those.

### 4.6.3 The muted resting state

An element whose range ends plays `exit` and goes. `after: "dimmed"` is the other resting state: the point stays and is drawn muted – legible, but no longer the thing being talked about.

```
#anim(at: "2-3", after: "dimmed")[A passing remark.]
#anim(at: 4)[And on with the talk.]
```

Nothing moves and nothing is recoloured: the element settles to 65 percent opacity and comes back up when you page back. `after` is `"hidden"`, the default, or `"dimmed"`.

`after` wants a range that ends **and** a step after it, or there is nothing to rest in and no step to be seen muted on; both are errors at compile time. `at: "3"` is that one step, `at: "2-3"` a range, and the second line above supplies the step after it. As a list, `at: (2, 4)` shows the element on 2, hides it on 3, brings it back on 4 and rests it dim from 5.

**On paper `after` does nothing**: a page shows every step at once, and a point that is only quiet because the talk moved past it has no past there.

> **ACHTUNG**
> The 65 percent keeps dimmed body text in the `ink` colour above 4.5 to 1 on every bundled palette. What is already quiet becomes too quiet: `muted` text or a word in the accent colour falls below that. Dim a point, not a label. Over a `card(fill: ...)` of your own or over an image nothing is measured at all.

A tracked element **inside** a dimmed one inherits the dimming only if it has exactly the same range – the same inheritance by which `enter`, `delay` and `duration` reach inwards. It may be **less** visible than its host, never more.

That leaves `morph`, `video`, `embed` and `flipbook` outside, because all four default to the open range `at: "1-"`. Inside a dimmed element they keep full strength, so a formula in a dimmed line stands black in a grey sentence. Give it the same closed range by hand, or do not dim the line.

## 4.7 Several versions in the same place

`alternatives` puts versions on top of one another. Each step shows exactly one, the next replaces it:

```
#alternatives(
  $ (a + b)^2 $,
  $ a^2 + 2 a b + b^2 $,
  $ a^2 + 2 a b + b^2 = c^2 $,
)
```

The box is as large as the largest version, so nothing around it jumps. `align` decides where the smaller ones sit inside it, `start` on which step the first appears, and `inline: true` puts the whole thing in a line of text. `enter`, `duration` and `easing` describe the change from one version to the next.

`morph: true` is the other way, and for the example above it is the better one: the versions fly into one another instead of replacing one another. They stand in the same place, so the flight has no distance – what you see is the glyphs rearranging themselves where they stand, which is what a rewritten formula does. With `morph` there is no entrance, so `enter:` and `easing:` are refused; `duration:` becomes the time of the flight.

## 4.8 A drawing that grows

A CeTZ canvas and a lilaq diagram are **one** piece, not many: Typst hands out the finished setting, and a line or a data series in it cannot be reached from outside. So there is no `anim` around a single line of a drawing – there is the drawing itself, as often as you want it. `build` calls it once per step and lays the versions exactly on top of one another: on stage $k$ the drawing stands as it looks after $k$ steps, and exactly one is on show.

Which piece joins when is said by the question every stage is handed. It is called `ab` – „from“ – because it says what `at:` says elsewhere:

```
#build(from => cetz.canvas({
  import cetz.draw: *
  line((0,0), (4,0))                          // there from the start
  line((4,0), (4,3), stroke: from(2, black))    // from step 2
  line((4,3), (0,0), stroke: from(3, 1.4pt + red))
  content((2.2, 1.8), from(4, [$c$]))
  if from(4) { circle((4,0), radius: 0.18) }
  else { hide(circle((4,0), radius: 0.18), bounds: true) }
}), steps: 4)
```

`from(2, black)` gives the colour back once the second piece is due, and otherwise the same colour with alpha 0. What carries no number stands there from the start. `steps: 4` says how many stages there are; it is said, not guessed, because nobody can see from outside what the drawing function does with its question. `from(4)` with a single argument is the same question as a boolean, for everything that cannot be recoloured – in CeTZ that is where `hide(…, bounds: true)` belongs.

Air rather than omission, because a piece left out takes its room with it and the drawing jumps. `from` makes air out of a colour, out of a stroke (the brush goes, thickness and dashing stay, because the measure hangs on those), out of the colours in a dictionary, and out of content, which goes into `hide`. Not out of a gradient – there it says so instead.

### 4.8.1 A lilaq diagram

A data series turns to air in two places: at its colour and at its label in the legend. The second is easy to forget – the entry would otherwise stand in the legend while its curve is missing:

```
#build(from => lq.diagram(
  width: 7cm, height: 4.5cm,
  legend: (position: top + left),
  lq.plot(x, measured, color: from(1, red), label: from(1, [measured])),
  lq.plot(x, model, color: from(2, blue), label: from(2, [model])),
), steps: 2)
```

Because the series stays in the data as air, lilaq reckons its axes over both: the scale is settled from the start, and the first curve does not jump when the second arrives.

### 4.8.2 The arguments

| Argument | Effect |
|---|---|
| `steps` | number of stages, and hence of steps (default 2) |
| `start` | first step; `auto` follows on from the cursor |
| `enter` | motion a stage arrives with (default `"fade"`); `"draw"` is an error here, see the next section |
| `duration` | duration in milliseconds |
| `easing` | the curve of the motion, see „The curve“ |

On paper only the last stage is set, in a block of the same size. Under „reduce motion“ nothing changes: the stages fade, they do not travel.

> **ACHTUNG**
> Every stage really is typeset. Four stages mean four layouts and four SVG trees in the file – for an elaborate drawing both grow as fast as they do for a flip book. A drawing in twenty stages is not a good idea.

## 4.9 A path that draws itself

`enter: "draw"` lets a stroke **come into being** instead of fading in: the pen is set down and traces the path from start to end.

```
#anim(circuit, enter: "draw", duration: 900)
#stagger(enter: "draw", stride: 1, axes, curve, tangent)
```

Behind it lies `stroke-dasharray` on the SVG path: one dash exactly as long as the path, slid in by `stroke-dashoffset`. `duration` applies as everywhere, but a drawing wants more time than a bullet point – 900 is a workable start, and the presentation's default of 520 is tight for three long lines.

### 4.9.1 What can be traced and what cannot

**Text cannot.** Typst sets glyphs as filled shapes with no outline, and an area has no length to travel along – the same for an arrow head, a solid dot, the face of a card. So `draw` does two things at once: **the strokes draw themselves, everything else fades in**, over the same time. A label arrives while the lines are being drawn and stands finished together with them.

An element on which **nothing at all** can be traced fades in completely, and the runtime says so in the browser's console, once per element:

```
typstage: enter: "draw" on slide 4 (element 2) finds no stroked path to
trace. What is drawn is an outline, and text has none: Typst sets glyphs
as filled shapes. The element fades in instead. draw is for a drawing,
the fade is for text.
```

It cannot be caught earlier: Typst hands out the SVG only on export, so only in the browser is there a path to count.

### 4.9.2 All at once, and how to get them one after another

Every stroked path of an element sets off **at the same time**, and there is no knob for that: the order in the SVG is Typst's painting order, not one the deck chose. Say the order instead, by giving each piece its own step:

```
#stagger(enter: "draw", stride: 1, axes, curve, tangent)
```

### 4.9.3 Where a drawing has to stand

**Not on the first step of its slide.** Entering a slide plays no entrances – the runtime only restores the state, or the transition and a dozen reveals would run against each other. A drawing on step one would simply be there. Give it a step in front:

```
#anim[First the sentence that announces the drawing.]
#anim(circuit, enter: "draw", duration: 900)
```

That holds for every effect; with `draw` it merely stands out, because there the travel is the whole point.

### 4.9.4 Who delivers outlines

**Whatever gets a `stroke` in Typst becomes a path with an outline and can be traced; whatever gets a `fill` does not.** A drawing package delivers exactly as much as it strokes, and a slide of text delivers nothing.

That decides between `draw` and `build`. A plain CeTZ drawing strokes a handful of paths – a few long lines an eye can follow, which is what `draw` was made for. A lilaq diagram strokes nearly everything, grid, ticks and markers included, and all of it sets off at once: a diagram wiping in, not a drawing coming into being. For a diagram, use `build`.

**Dashed lines stay with the fade.** The dash pattern lives in the very attribute the pen needs, so a dashed guide line fades in while its neighbours draw themselves.

### 4.9.5 In both directions, and what holds at the edges

| Where | What happens |
|---|---|
| Paging back | The pen traces its way out: what drew itself undraws itself. |
| Jumping to a step | No drawing. A jump – address, overview, reload – restores the end state, which is the finished drawing. |
| `exit: "draw"` | Allowed and symmetric: an element leaving its range takes its strokes back instead of fading away. |
| Speaker view | The preview of the next step shows the finished drawing, with no motion. |
| Paper | Nothing. `enter` never reaches the PDF. |
| Reduce motion | The pen holds still, the fade remains – **opacity stays, travel goes**, and for `draw` the drawing **is** the travel. What is left is the fade that ran underneath it anyway, over the same duration. The console message still comes. |

### 4.9.6 Together with a drawing that grows in stages

Both at once does not work, and the package says so at compile time:

```
#build(painter, enter: "draw")   // an error at compile time
```

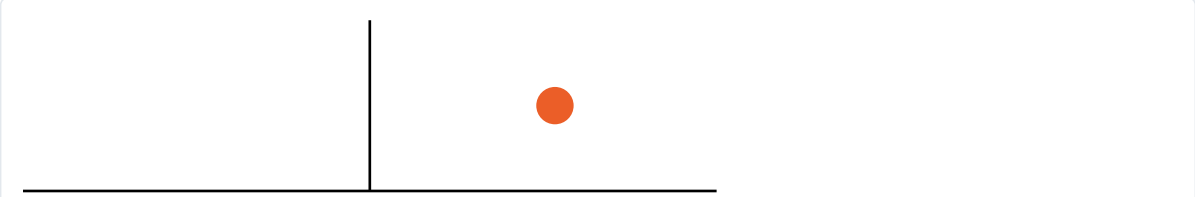
Every stage of a `build` drawing is the **whole** drawing, so a stage that drew itself would retrace every stroke over ink already down. For strokes that come into being one by one, hand them over as pieces of their own; for a diagram that grows in stages, leave it with its fade.

## 4.10 A drawing that moves

`build` lets a drawing grow, piece by piece. `scene` is the other half: nothing is added, a **value** changes, and the picture hangs on it.

**The deck writes a function from a value to a picture and names the values at which the talk stops. Typst renders every stop and the frames in between, and a step pulls the picture from one stop to the next.**

```
#scene(
  x => drawing-at(x),
  stops: (-3, 0, 1.5, 3),   // four stops, three steps
  tween: 8,                 // frames between two stops
)
```



`stops` are the values themselves, not `0.0` to `1.0`. That is the difference to the flip book: there `t` is a fraction of a running time, here `x` is the quantity being talked about. Whoever wants the tangent at $-3$, at the vertex and at $1.5$ writes those three numbers down.

The scene takes `stops.len() - 1` steps. The first stop is there as soon as the scene appears – like a `morph`, unlike an `anim` – and every further stop costs a keypress.

### 4.10.1 What belongs to a stop

A sentence, a formula, a second drawing: `scene-layer` puts itself on the step of one particular stop. The scene needs a name to be found by.

```
#scene("derivative", x => tangent-at(f, x), stops: (-3, 0, 1.5, 3))

#scene-layer("derivative", 2)[At the vertex the slope is zero.]
#scene-layer("derivative", 4, enter: "scale")[$f'(x) = 1/2 x$]
```

This is word for word `cue-layer`: the coupling falls out of the shared step. Move a stop and everything hanging on it moves along, and nowhere does a number stand twice. The scene has to stand **before** its layers.

### 4.10.2 Several values at once

A stop may be a tuple, and then the drawing function takes that many arguments:

```
#scene(
  (a, b) => box-of(width: a, height: b),
  stops: ((1, 1), (1, 3), (2, 3)),
```

```
  tween: 6,
)
```

First the height grows, then the width. What does not work: two values moving **independently**. Everything travels from stop to stop together, and a tuple puts several values on the one way.

### 4.10.3 The arguments

| Argument | Effect |
|---|---|
| `stops` | The values at which the talk stops. At least two. A number, a length, an angle, a ratio – or a tuple of them. |
| `tween` | Frames **between** two stops (default 8). With `0` the scene jumps. |
| `start` | first step; `auto` takes the running one |
| `width`, `height` | The box the scene stands in (default `100%` and `190pt`). |
| `duration` | how long one pull from stop to stop takes, in milliseconds |
| `enter` | motion the scene itself arrives with (default `"fade"`) |
| `still` | what stands on paper, if not the last stop |
| `steady` | What measuring the frames is for: `auto` reports, `false` takes the scene out of the check, `true` insists on it. See below. |

`duration` is the duration of the **journey**, not of the fade the scene arrives with. Unlike `build`, `scene` does not stack its frames: they are drawings of different values and may legitimately come out different sizes. So a scene stands in a box of fixed size, every frame is clipped to it – and every frame is measured:

> ACHTUNG
> **The box stands still, the ink inside it does not do so by itself.** A CeTZ canvas grows with its content, so if the tangent at $x = -3$ reaches further left than the one at $x = 3$, the axis cross sits elsewhere in the box, and paging moves the whole picture although only one point was meant to move. Every scene measures its frames and says so where the sizes differ:
>
> ```
> error: assertion failed: typstage: 1 scene draws frames of different sizes. …
>   slide 4, from step 1: 28 frames in 19 different sizes, up to 28.35pt apart across and
> 53.86pt down
> ```
>
> The way out lies in the drawing: give it a fixed extent and keep what moves inside. In CeTZ that is a `rect` with a transparent stroke:
>
> ```
> #scene(x => cetz.canvas({
>   import cetz.draw: *
>   // Holds the canvas open, wherever the point stands.
>   rect((-4.4, -0.8), (4.4, 4.6), stroke: rgb(0, 0, 0, 0))
>   line((-4, 0), (4, 0))
>   circle((x, 0.25 * x * x), radius: 0.1)
> }), stops: (-3, 0, 3), height: 160pt)
> ```
>
> That pins the width. Whatever still reaches beyond it – a tangent running off the edge – has to be cut off, or it pulls the canvas open again.
>
> **Where the frames are meant to differ**, say so: `steady: false` takes the scene out of the check. `drift` on the presentation decides what happens with the findings.

`steady: true` is the opposite commitment: the scene has to stand still, and it stops on the spot rather than in a list at the end of the deck:

```
#scene(x => cetz.canvas({
  import cetz.draw: *
  line((0, 0), (x, 0.25 * x * x))            // pulls the canvas along
}), stops: (-3, 3), steady: true)            // error at compile time
```

On paper the last stop is set, as with `alternatives`; `still` puts something else in its place. The step cursor runs there too, so `info().step.total` names the same number in both outputs. Under „reduce motion“ the frames in between fall away and the scene jumps.

### 4.10.4 What a scene costs

Every frame really is a Typst layout and sits in the file as an SVG tree of its own, so compile time and raw file size grow with `tween`. Over the wire it matters far less: the trees are so alike that gzip takes some 98 percent away. On paper a scene costs nothing – one still image. Measuring the frames costs one more layout each, in the browser branch only; `steady: false` gives it back for one scene, `drift: "none"` for all.

> **ACHTUNG**
> The gzipped figure holds only as long as the web server does gzip; whoever hands the file on by USB stick or as an attachment carries the raw one. And the compile time is always the full one: eight frames per stretch are eight layouts, whether they compress away later or not.

## 4.11 Moving in on a detail

Sometimes the next step is not a new sentence but the same one from close up: the one cell of the table, the one term of the equation. `camera` moves in on it and back out again. It aims at a `pin` and at nothing else – the package's word for a named piece of a slide, whose rectangle the runtime measures anyway.

```
#pin(<sensor>, card(title: [Sensor])[Thermocouple, bridge, amplifier.])

#camera(<sensor>)
#anim[And out again, on the step after.]
```

### 4.11.1 How you get out again

Said, not guessed. `at` is a step selector as everywhere else, and the slide is seen through the camera for as long as it is active:

| Written | What happens |
|---|---|
| `at: auto` | The next free step, and the one after takes it back out. The default. |
| `at: "3"` | In on step three, out on four. |
| `at: "3-5"` | The crop holds across three steps. |
| `at: 3` | In on step three and stay; the slide change takes it out. |

The way back out is a step and is counted as one: a slide carrying nothing but a pin and a camera has three steps – the whole slide, the crop, the whole slide. `info().step.total` and the handout count the same way.

> **HINWEIS**
> `at: auto` is a **closed** range here, while for `anim` it is open: an entrance has no natural end, a camera move does. And never step one – a move there would mean nobody ever saw the slide whole.

### 4.11.2 What travels along and what stays put

What travels is the slide: background and the layer of revealed parts above it, with the same transform. The furniture does **not** – footer, page number, progress and running header sit as their own layer above the stage, hold still while the slide grows underneath them, and stay legible. The title travels; it stands in the body. What leaves the frame is cut at the edge of the stage, and drawn ink stays put.

### 4.11.3 How far it goes

`margin` says how much of the slide stays around the detail, measured on the **unzoomed** slide (16 pt by default). The camera fits detail plus margin into the frame, and the tighter direction decides, so the whole of it is seen. The move takes `duration` milliseconds, 700 by default, and `easing` bends it.

```
#pin(<term>, $b^2$)
#camera(<term>, margin: 4pt, duration: 900, easing: "out-quad")
#anim[After that.]
```

There is no upper limit. A pin the size of a comma is shown the size of a wall, and what Typst set stays sharp, because it stands there as vectors; a video, an image or an embedded document will not. A detail already as large as the slide gives nothing to travel to.

### 4.11.4 Two special cases

**Two pins of the same name on one slide.** The camera frames the box around both.

**Two moves overlapping on one step.** The later one in the source wins.

### 4.11.5 On a jump, paging back, and on paper

The crop is a function of the step and nothing else:

- **Paging back** runs the way in reverse and lands on the whole slide again.
- **A jump** – overview, `#3` in the address, a click in the speaker view – sets the crop instead of travelling to it.
- **The speaker view** shows the running slide with its camera, and the preview beside it carries the crop along: its question is „what stands there after the next keypress“.
- Under **reduced motion** the camera jumps to the crop.

> ACHTUNG
> **On paper there is no camera.** The handout sets every slide whole, and so does the browser's print view. A duty follows: **the slide has to be complete and legible without the move.** Whoever labels the detail only for the crop – a 6-point line, since we are going to move in on it anyway – has a line on paper that nobody reads. The camera is an emphasis, not a layout.

### 4.11.6 When the name is not there

A camera aiming at a `pin` that does not exist on its slide is an error at compile time:

```
#pin(<sensor>, card[…])
#camera(<senor>)            // one letter short
```

The question is asked at the end of the document, not on the spot: a move may stand before its target, and what stands on a slide is only settled once the slide is set. A pin on the slide **before** does not count.

One case stays open: a pin inside an `anim` not revealed on this step has a rectangle but nothing visible in it, and the camera moves in on an empty place. Which step shows what is decided in the browser.

## 4.12 Three stumbling blocks

**Only reveals count.** The cursor counts `anim`, `stagger`, `alternatives` and `#pause` – everything that makes something appear. An applet, a video or a `morph` uses up **no** step and is there from the beginning. That matters in a two-column slide: the bullets beside an applet should start at one, not behind its motions.

```
#side-by-side(
  embed(url: "…", width: 100%, height: 220pt),   // no step
  stagger[
    - first bullet                               // step 1
    - second bullet                              // step 2
  ],
)
```

**A step is not inherited inwards.** Every tracked element carries its own step, and one sitting inside another still follows it:

```
#anim(at: 3)[From step three, #morph(<m>, $x^2$) but from step one.]
```

With `morph` that is right: the **target** of a flight has to be standing when the slide is entered, or the flight from the previous slide arrives nowhere. With an `anim` inside an `anim` it is usually an oversight, noticed only while paging.

**A morph stands from the first step.** So a morph does not belong inside something that only appears later. Put it in a tile that arrives on step two and it hovers alone on step one, where its container will only later turn up.

# 5 Showing instead of claiming

Three ways to make a slide demonstrate something rather than assert it, from the most involved to the simplest.

## 5.1 A document of your own on the slide

`embed` puts arbitrary HTML into a sandboxed frame:

```
#embed(html: "<div id=lamp></div><script>…</script>",
       width: 100%, height: 190pt)
```

`url:` takes a foreign address instead.

> **TIPP**
> Size everything inside in `em`. Inside a zoomed frame one CSS pixel is one point of the slide, so `em` scales with the slide and `px` does not. A page that reflows on its own wants `zoom: false`.

`style: false` drops the deck's basic style where the embedded document brings its own. `fallback` stands on paper in the frame's place, `link` is the address printed beneath it.

## 5.2 Sending it something on a step

A frame with a `bridge:` argument gets a name; `bridge-job` sends it a dictionary when a step arrives:

```
#embed(html: "…", bridge: "lamp", width: 100%, height: 190pt)

#bridge-job("lamp", (color: "#16a34a"), at: 2)
#bridge-job("lamp", (color: "#eb5e28"), at: 3)
```

The package never reads a job; the document on the other side interprets it. That is how the `ggb-` commands drive their applets — see the chapter **GeoGebra**.

> **ACHTUNG**
> The document has to announce itself once with `postMessage({typstage: 1, ready: 1})`. Until it does, it gets nothing.
>
> Paging back replays the whole run with a `reset`, so a job has to be repeatable: „set the colour to green“ survives that, „make it greener“ does not.
>
> Two frames sharing a name both receive every job, and the runtime says so in the console. `bridge-targets()` reports the names on the current slide.

## 5.3 Video

```
#video("clip.mp4", width: 100%, height: 260pt, poster: image("still.png"))
```

The file travels beside the HTML, not inside it. `autoplay`, `loop`, `muted` and `controls` are the usual switches; `poster` stands there before it runs and takes its place in the PDF. The frame crops rather than stretches.

## 5.4 A flip book

`flipbook` lets Typst render the motion itself, frame by frame:

```
#flipbook(
  t => box(width: 100%, height: 100%,
    place(left + horizon, dx: t * 88%, circle(radius: 9pt, fill: accent))),
  frames: 24, fps: 20, width: 100%, height: 46pt,
)
```

The function receives `t`, running from 0 to 1, and is called once per frame. It can draw with anything Typst has, CeTZ and Fletcher included, and every frame sits in the file as SVG. This is the tool for motion Typst can draw and CSS cannot: a traced curve, a turning mechanism.

`loop`, `pingpong` and `still` decide how it plays and which frame stands on paper. A viewer who has asked for „reduce motion“ never sees it play.

The clock starts when the flip book becomes visible: `flipbook(at: "3-")` lies on frame 0 until step 3, then plays from zero – again on every fresh reveal.

> **ACHTUNG**
> Every frame is typeset separately: twenty-four frames are twenty-four layouts and twenty-four SVG trees in the file. This is the most expensive element in the package. Reach for it only where the motion carries the argument.

# 6 GeoGebra

GeoGebra builds the construction, the slides supply the dramaturgy. A job can sit on every step: set values, show or hide objects, change colours, move the viewport, start a motion.

## 6.1 Quick start

`geogebra()` puts an applet on the slide. The commands that drive it stand in the same slide body and produce no output of their own.

```
#import "@preview/typstage:0.1.0": *

#presentation(
  slide([Remote controlled], {
    geogebra(app: "classic", perspective: "G", height: 240pt,
             link: "https://www.geogebra.org/calculator")
    ggb-run("a=1", "f(x)=a*x^2")
    ggb-set((a: 3), at: 2)
  }),
)
```

The parabola is there from the start; on step 2 `a` becomes 3.

Every command takes `at`, the step selector known from `anim`; the default is `"1-"`. The applet frame has no step of its own, so bullet points beside it still start on step one.

> **HINWEIS**
> The applet lives in the HTML export only; for the PDF see *On paper*.

## 6.2 Which applet is meant

With one applet on the slide the commands find it themselves. Two applets need names, and the commands then need `target` — a string or a label:

```
#geogebra(<left>, height: 200pt)
#geogebra(<right>, height: 200pt)
#ggb-run("A=(0,0)", target: <left>)
#ggb-run("B=(1,1)", target: "right")
```

With no applet, or with more than one and no `target`, nothing is guessed. The build stops and names what it found:

```
error: panicked with: typstage: 2 applets on this slide
(left, right) — say which one is meant, e.g. target: "left".
```

## 6.3 Building the construction

`ggb-run` hands GeoGebra commands to `evalCommand`, one at a time. The order counts: whatever is needed has to exist first.

```
#ggb-run(at: "1-",
         "k: x^2+y^2=4", "t=Slider(0,6.283,0.01)",
         "P=(2cos(t),2sin(t))", "s=Segment((0,0),P)")
```

> **ACHTUNG**
> GeoGebra's scripting commands — `SetColor`, `SetValue`, `SetVisibleInView` and their relatives — are **not** accepted by `evalCommand` and come to nothing inside `ggb-run`. Use `ggb-set`, `ggb-style`, `ggb-show` and `ggb-hide` instead. Rejected commands land in the browser's console.

Entering a slide and paging back reset the applet and repeat the run, so commands have to be repeatable. Fix the colour on `"1-"` for the same reason: on a rebuild GeoGebra would otherwise hand out the next colour of its palette.

```
#ggb-run("a=1", "f(x)=a*x^2", at: "1-")
#ggb-style("f", at: "1-", color: dark, thickness: 3)
```

> **HINWEIS**
> A `.ggb` file cannot be embedded: Typst has no way to inline binary data into the HTML. Build the construction with `ggb-run`, or load it from GeoGebra through `material: geogebra(material: "abc123xy")`.

## 6.4 Values, appearance, viewport

`ggb-set` takes a dictionary of object name and value, `ggb-show` and `ggb-hide` any number of object names. Build everything at the start and reveal it when its turn comes:

```
#ggb-hide("P", "s", "t", at: "1-")
#ggb-show("P", "s", at: 2)
#ggb-set((a: 3), at: 2)
#ggb-set((a: -2, b: 0.5), at: 3)
```

### 6.4.1 Appearance

`ggb-style` takes the object names and the settings to change. What is not named stays as it is.

| Setting | Effect |
|---|---|
| `color` | colour, as a Typst colour and not a GeoGebra one |
| `thickness` | line weight |
| `line-style` | line style as a number (solid, dashed, dotted …) |
| `filling` | fill, 0 to 1 |
| `point-size` | point size |
| `trace` | trace on or off |
| `label` | label visible or not |
| `label-mode` | kind of label as a number (name, value, caption …) |
| `fixed` | held against being moved |
| `caption` | a caption of your own |
| `layer` | layer, that is, what lies in front of what |
| `position` | place as `(x, y)` |

`color` takes a Typst colour, so the construction carries the colours of the slides instead of GeoGebra's palette.

```
#ggb-style("P", at: 2, color: accent, point-size: 6)
#ggb-style("s", at: 2, color: dark, thickness: 3)
#ggb-style("d", at: 3, color: accent, filling: 0.18, thickness: 4)
```

**ACHTUNG**
`position` counts in coordinates of the plane, except for a slider made with `Slider`: that one sits at an absolute place on the screen and counts in pixels. Two sliders both written as `(-3.9, 2.2)` land in the same corner.

### 6.4.2 Viewport

`ggb-view` sets the visible range as well as the grid and the axes. `x` and `y` take effect only together; each is a pair of smallest and largest value.

```
#ggb-view(at: 2, x: (-3, 3), y: (-3, 3), grid: false)
#ggb-view(at: 3, axes: false)
```

**ACHTUNG**
`ggb-view` sets x and y separately, so a range that does not match the shape of the box stretches one axis and a circle becomes an ellipse. Where the geometry carries the argument, give the box a fixed size and match the ranges to its proportions.

Without `ggb-view` the visible range follows from `width` and `height`.

## 6.5 Motion

Two ways to set something moving, and they do different things.

`ggb-animate` starts GeoGebra's own animation: back and forth without end until the slide is left. `trace` switches on the trace of the named objects, `speed` sets the pace, `playing: false` stops it.

```
#ggb-animate("t", at: 3, speed: 1.2, trace: ("P",))
```

`ggb-tween` moves a value once from A to B and stops. Everything that depends on it follows along — a segment whose endpoint travels, an arc whose angle grows — and that is how a construction draws itself. `from` gives the starting value, `duration` the time in milliseconds, `easing` the shape.

```
#ggb-run("t_1=0", "s=Segment(A,(4*t_1,0))", at: "1-")
#ggb-tween("t_1", at: 2, to: 1, duration: 700)
```

**ACHTUNG**
`ggb-tween` needs a step number, not a range: `at: 2`, not `at: "2-"`. Otherwise the build stops with „`ggb-tween() needs a step number`".

A tween on step 1 never arrives as motion: on entering a slide the runtime replays the run up to the current step at once, and tweens jump to their target value. Step 1 builds up, drawing starts at step 2.

From the next step on the value sits on its target, so paging back shows the finished drawing instead of the motion again.

## 6.6 On paper

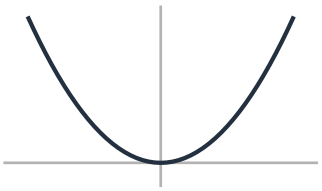
There is no applet in the PDF. A labelled placeholder keeps the size of the frame, and `link` puts the way to the live applet beneath it.

```
#geogebra(height: 90pt, link: "https://www.geogebra.org/calculator")
```


https://www.geogebra.org/calculator

Better is a drawing of your own. `fallback` takes any content: an image, a table, above all a drawing with CeTZ.

```
#geogebra(height: 120pt, link: "https://www.geogebra.org/calculator",
  fallback: cetz.canvas(length: 0.8cm, {
    import cetz.draw: *
    line((-2.6, 0), (2.6, 0), stroke: luma(70%))
    line((0, -0.4), (0, 2.6), stroke: luma(70%))
    line(..range(0, 45).map(i => (-2.2 + i * 0.1, 0.5 * calc.pow(-2.2 + i * 0.1, 2))),
         stroke: dark + 1.6pt)
  }))
```



https://www.geogebra.org/calculator

> **TIPP**
> Where the applet runs through several states, the better stand-in is the whole run as a row of pictures, not a photograph of one step.

Both take effect in the PDF only.

## 6.7 How the applet looks

`seamless: true`, the default, takes the frame off the applet and puts its drawing area in the colour of the slide. It then looks like part of the slide rather than a window inside a window. `background` sets that colour.

```
#geogebra(height: 240pt, background: rgb("#f4f1ea"))
#geogebra(height: 240pt, seamless: false)   // with GeoGebra's own frame
```

`background: auto`, the default, takes the paper of the theme in force, so an applet on a dark theme comes up dark.

> **ACHTUNG**
> The viewport cannot be dragged by hand, and that is the default: whoever reaches beside the point during a talk would otherwise push the whole plane away. `pan: true` gives dragging and zooming back; points and sliders can be dragged either way.

`font-size` counts in points of the slide, like `width` and `height`, so the applet's font grows with the slide instead of staying physically the same size on a projector. The default is 17, one above GeoGebra's 16.

> **ACHTUNG**
> GeoGebra snaps the font size to steps, so neighbouring values often come out at the same height.

```
#geogebra(height: 240pt, font-size: 22)      // larger axis numbers
#geogebra(height: 240pt, pan: true)          // viewport by hand
```

`grid` and `axes` follow GeoGebra's own default while they are `auto` and force one or the other otherwise. `perspective: "G"` shows the graphics view alone, `app` chooses the GeoGebra app (default `"classic"`), `language` the interface language, and `animation-button` shows GeoGebra's play button.

### 6.7.1 Size

`width` and `height` count in the measurements of the slide, not in screen pixels, and `width: 100%` is the usual case. What keeps every window showing the same crop is the visible range: it is set from the box the first time the applet appears, and after that `ggb-view` decides.

> **TIPP**
> Two applets side by side sit best in a `grid`, each with `width: 100%` and a height of its own.

## 6.8 From the speaker view

The speaker window runs a copy of every applet. `m` switches its pointer from the pen to the embedded frame; the applet in front of you is then the live one, and the copy on the canvas follows what you do to it.

Only what a hand has touched travels: a dragged point, a slider, the panned view. Creating, deleting or renaming sends the whole construction. An animation running on both sides sends nothing.

> **ACHTUNG**
> A step change resets both copies and replays the jobs of the slide. A change made by hand lives as long as the step does. Where a position is meant to stay, it belongs in the deck with `ggb-set`.

> **TIPP**
> Pin down whatever is not meant to move: `ggb-style("A", "B", fixed: true)` nails the points that merely span a construction. Otherwise a hand in the talk easily takes the wrong one — with Thales, the diameter instead of the point on the half circle, and the whole arc travels with it.

`Point(k)` is a point on the path that a hand can take; `Point(k, 0.3)` is pinned to that parameter and cannot be dragged at all. Where it should start is said with `position:`. `examples/geogebra-sprecher.typ` is a deck built around exactly this: Thales with a point that walks along the half circle and leaves its trace, and a parabola with two sliders.

### 6.8.1 The keyboard

Click the applet and it holds the focus; every key then lands inside it. The keys the talk uses are handed back out of the frame — see „A frame that has the focus". Without a toolbar and without an algebra input, no key changes the construction anyway.

## 6.9 Whose applet this is

This package does not ship GeoGebra. The browser fetches what runs in the frame from `codebase`, `https://www.geogebra.org/apps/` by default. Three things follow:

1. **Without a network the frame stays empty.** Whoever presents offline puts GeoGebra's files beside the deck and points `codebase` at them.
2. **The applet stands under GeoGebra's terms**, not under this package's MIT licence, which covers the Typst and runtime code here. For commercial use, read GeoGebra's.
3. **The viewer's browser talks to `geogebra.org`.** Where that is unwanted — a firewall, a data protection requirement — `codebase` sends it elsewhere.

> **HINWEIS**
> On paper none of this is left: the PDF fetches nothing.

# 7 Desmos

The same road as GeoGebra, a different calculator. Everything the previous chapter says about the bridge – step selectors, choosing a target, repeatability – holds here unchanged; this chapter names only what differs.

The difference at the core is the language: GeoGebra takes commands, Desmos takes **expressions**. Each carries an `id`, and the same `id` again replaces the expression rather than adding a second one. That is how a curve moves across the steps.

## 7.1 The key

`api-key` is required. Desmos serves its script only against a key – without one the server answers 403 and the frame would stay empty.

For trying things out there is `demo-key`, which Desmos names for that purpose in its own documentation. It works, but the script says in the browser console where you stand:

```
This page is using the Desmos API with a trial key suitable for prototyping,
not for commercial use.
```

> **ACHTUNG**
> Giving a talk on that key means working outside what it is meant for. A key of your own comes from https://www.desmos.com/my-api. And it stands in the HTML in plain text – a deck you hand on or put online hands the key on with it.

## 7.2 Quick start

```
#import "@preview/typstage:0.1.0": *

#presentation(
  slide([A parabola], {
    desmos(api-key: demo-key, height: 300pt,
           expressions: (a: "a=1", kurve: "y=a x^2"),
           bounds: (-5, 5, -2, 12))
    dsm-set(("gerade": "y=2x"), at: 2)
  }),
)
```

`expressions` is the opening picture, `bounds` the viewport as `(left, right, bottom, top)`. On step 2 a line joins it.

An expression with `=` is a slider, one without is a curve; Desmos decides that, not this package.

## 7.3 Showing, hiding, removing

```
#dsm-hide("gerade", at: 3)
#dsm-show("gerade", at: 4)
#dsm-remove("gerade", at: 5)
```

The difference matters more than it looks: a hidden expression stays in the calculator and keeps computing, so whatever depends on it still holds. A removed one is gone, and everything that names it goes with it.

## 7.4 Appearance and viewport

`dsm-style` takes Desmos' own keys. `color` takes a Typst colour, so the curve can carry the palette of the slide.

```
#dsm-style("kurve", color: rgb("#eb5e28"), line-width: 3, at: 2)
#dsm-view(bounds: (-2, 2, -1, 4), at: 3)
```

`dsm-view` moves the viewport and switches grid, axes and axis numbers. What `desmos()` is given at build time holds at the start; what `dsm-view` sends holds from its step on.

## 7.5 Motion

Two ways, and they do different things.

`dsm-animate` switches on Desmos' own slider animation. It runs at Desmos' speed and without a destination, back and forth until someone stops it – right for a picture that should breathe.

`dsm-tween` pulls a slider from one number to another and leaves it there – right for a step that shows something.

```
#dsm-tween("a", to: 3.0, at: 2, duration: 900)
#dsm-animate("b", at: 4, min: -3, max: 3, step: 0.1)
```

> **ACHTUNG**
> On `dsm-tween`, `at` is a **step number** and not a selector, and that is deliberate. The motion sits on exactly that step; from the next one the command simply sets the end value. Were it „from step 2 on", it would start again on every further step of the slide – measured, the slider jumped back from 3 to 0.75 and grew again.

## 7.6 On paper

As with the applet next door: the calculator lives in the HTML only. The PDF shows what `fallback` says, and without a `fallback` the space stays empty. A picture of the finished graph is usually the better answer there than an empty box.

## 7.7 Whose calculator this is

The frame fetches Desmos from `desmos.com` when the slide is shown. Without a network it stays empty, the viewer's browser talks to that host, and what runs inside is under Desmos' terms rather than this package's MIT licence. The PDF fetches nothing.

A deck that never calls `desmos` carries none of it: boot script and frame document sit behind the call.

# 8 Developing a calculation

An equation that rewrites itself in front of the room instead of being replaced by the next one.

## 8.1 One name, two slides

The same name on two slides, and the thing flies across:

```
== Step 1
#morph(<term>, $ (a + b)^2 $)

== Step 2
#morph(<term>, $ a^2 + 2 a b + b^2 $)
```

The name is a string or a label; the runtime pairs the glyphs at both ends and moves each one to its new place.

## 8.2 And on one slide

A morph flies between two steps, and two steps of one slide count for as much as two slides. Use two calls of the same name with ranges that do not overlap:

```
== Completing the square

#statement[#morph(<sq>, $ x^2 + 6 x $, at: "1")]
#statement[#morph(<sq>, $ (x + 3)^2 - 9 $, at: "2-")]

#anim([And one step, so that there is a second one.], at: "2-")
```

A morph takes no step of its own, so the slide needs a second step from somewhere else – an `anim`, a `stagger`, anything.

The name also has to be free on the slide before: a morph that starts after step one may not share its name with one on the previous slide. The package says so while compiling.

> **TIPP**
> Two versions in the same place fly no distance at all, and all you see is the glyphs rearranging themselves – often exactly right. To see movement, put the two versions one above the other.

## 8.3 Two shorthands for the common case

`alternatives(morph: true)` lets its versions fly into one another instead of replacing one another:

```
#alternatives(morph: true,
  $ (a + b)^2 $,
  $ (a + b)(a + b) $,
  $ a^2 + 2 a b + b^2 $,
)
```

`stagger(morph: true)` is the chain where every line stays: the new line grows out of the line above, which stays put.

```
#stagger(morph: true, spacing: 14pt,
  $ x^2 + 6 x + 2 = 0 $,
  $ (x + 3)^2 - 7 = 0 $,
```

```
  $ x = -3 plus.minus sqrt(7) $,
)
```

Both take a name of your own instead of `true`. That is needed only where the flight carries on past the edge of the slide.

> **ACHTUNG**
> A morph has no entrance, so both refuse `enter:` and `easing:` rather than quietly dropping them, and `stagger` also refuses `dim:` – an argument `alternatives` does not have. `duration:` is read, and it is the time of the flight.

## 8.4 How the pairing works

`match: "auto"` compares the outlines: two glyphs of the same shape find each other, and where that is not enough, proximity decides. `"glyph"` forces it per glyph, `"block"` moves the whole thing as one rectangle.

> **TIPP**
> `"block"` is the right answer more often than it looks. A picture or a table has no glyphs worth pairing, and per-glyph matching there gives a swarm rather than a movement.

Source order decides what lies *on top*, at rest and in flight: what is written after the `morph` lies above it. A caption need not wait for the picture to land.

## 8.5 When the wrong signs fly

Where the pairing goes astray, name the pieces. Matching `pin` names find each other before the shape is consulted:

```
#morph(<term>)[$#pin(<factor>)[3] x^#pin(<power>)[4]$]
// and on the next slide
#morph(<term>)[$#pin(<power>)[4] dot #pin(<factor>)[3] x^3$]
```

A pin without a counterpart on the other slide falls back to shape matching without complaint.

## 8.6 Duration and the first link

`duration` is 900 ms rather than the presentation's, since a flight takes longer than a fade-in; `auto` falls back to the presentation's value.

A morph is present from the first step, at both ends of a chain, since paging back swaps the roles. Only the **first** link may be delayed, because no flight arrives there; the package checks that at compile time.

## 8.7 Where the magic move stops

Two targets on the **same** slide may share a name. Both then start from the same place, and the glyph visibly splits in two.

> **ACHTUNG**
> A morph is typeset a second time, in a frame of its own, and that frame never sees a `#set` rule written in the document. Shared typography belongs in `style:` on `presentation`. This is the most common reason for a flying equation in the wrong font.

## 8.8 How the slide itself changes

`transition` decides how a slide comes in. The presentation sets the default, and a single slide may differ:

```
#show: presentation.with(transition: "slide", transition-duration: 420)

== This one differently
#transition("cover", from: "bottom")

// or, in the argument form:
#slide([This one differently], transition: (kind: "cover", from: "bottom"))[…]
```

| Kind | Takes | What happens |
|---|---|---|
| `"none"` | – | A hard cut. |
| `"fade"` | – | A cross-fade, nothing moves. |
| `"slide"` | `from` | The new slide moves in a short way and fades up while doing it, the old one gives way in the other direction. |
| `"push"` | `from` | The new one pushes the old one over the edge. |
| `"cover"` | `from` | The new one lays itself over the old one, which stays. |
| `"uncover"` | `from` | The old one moves away and frees the new one. |
| `"zoom"` | `direction` | `"in"` grows the new one forward, `"out"` steps the old one back. |
| `"blur"` | – | Out of focus and back. |
| `"iris"` | `direction` | A round aperture: `"open"` opens the new slide, `"close"` closes over the old. |
| `"wipe"` | `direction`, `from` | The same as a straight edge; `from` names the edge it starts at. |
| `"flip"` | `axis` | Turning over in space, like a leaf. |
| `"cube"` | `axis` | Like `flip`, but as two faces of a cube that keeps turning. |

`from` is `"right"` (the default), `"left"`, `"top"` or `"bottom"`. `direction` is `"in"` / `"out"` for `"zoom"` and `"open"` / `"close"` for `"iris"` and `"wipe"`, the first value being the default in each case. `axis` is `"y"` (the default, turning about the vertical) or `"x"`.

**The transition belongs to the boundary between two slides, not to the direction of travel.** What counts is the setting of the later slide; backwards it runs mirrored rather than again.

**Where a morph meets the slide, it cross-fades.** Otherwise the slide would push away the very object flying across it. A chain of transformations therefore needs no transition switched off by hand.

# 9 Giving the talk

Everything that happens between opening the file and the last slide, including the second window.

## 9.1 The keys

| Key | What it does |
|---|---|
| `→` `space` `PageDown` | one step forward |
| `←` `PageUp` | one step back |
| `Home` `End` | to the first or the last step |
| `o` `Esc` | the overview, and a click there goes to that slide |
| `f` | full screen |
| `?` | every key |
| `n` | open the speaker view, or bring the talk forward |

A click pages forward, a click in the left quarter pages back. The address bar carries the running step, `#12` being the twelfth, so a reloaded window stands in the same place and a number typed by hand jumps there.

### 9.1.1 A frame that has the focus

Click an embedded frame and it holds the focus: every key then lands inside it and the talk stops paging. The talk's own keys are therefore handed back out of the frame, under three conditions – the embedded document has not already taken the key, the key is one the talk uses, and the focused element is not a text field. Otherwise an `n` typed into a form would open a second window.

Everything else stays with the frame. `Delete` is the example: it belongs to whatever is embedded, and the talk never sees it.

## 9.2 On a phone or a tablet

A tap pages, in the same two halves as a click. A swipe pages in the natural direction: the finger pushes the slide out to the left, so the next one comes. Vertical swipes and two fingers are left to the browser: one is scrolling and the other is zooming.

## 9.3 The speaker view

`n` opens the same file a second time, with `#speaker` on the address, in a second window: one for the projector, one for the machine in front of you. The two talk over `postMessage`, which works between two local files, so no server is needed.

The view is a lectern made of tiles. Two large ones on top – the running slide on the left, the note on the right – and below them a row of four small tiles and one wide one:

| Tile | What stands in it |
|---|---|
| elapsed | the time since the first keypress, and small below it the time of day |
| slide | slide / slides, below it step / steps, and the progress bar along its foot |
| target (min) | the planned duration – `d` goes into it –, below it remaining and pace, once one is set |

| Tile | What stands in it |
|---|---|
| class clock | the clock that stands on the wall in the hall; `t` starts it |
| next step | the preview: what the next keypress does |

Below that the tool row: pen or pointer, the four colours, the key help. The state of the hall – `black`, `frozen`, `no talk window` – stands at the top right inside the slide tile.

The keys of the view, which `?` also shows inside it:

| Key | What it does |
|---|---|
| `←` `→` | one step; `Home` `End` to the first or the last |
| `↑` `↓` | scroll the note |
| `o` | the overview |
| `b` | black out the hall |
| `e` | freeze the hall on this step |
| `n` | bring the talk window forward |
| `t` | the class clock, full screen in the hall |
| `⇧t` | the same clock, but on the slide instead of over it |
| `⇧←` `⇧→` | one minute less or more, while a clock runs |
| `d` | the target duration in minutes |
| `r` | set the elapsed time back to zero |
| `m` | switch between pen and pointer |
| `c` | the next drawing colour |
| `z` | take back the last stroke |
| `x` | clear the strokes on this slide |
| `l` | light or dark, for the view alone |
| `+` `-` | the size of the note |
| `f` | full screen |
| `?` | this table, in the view |

### 9.3.1 What the view should show

`speaker-view` cancels what is not wanted, so an unused tile does not take room the note could use:

```
#show: presentation.with(speaker-view: (
  clock: false,                                  // no class clock
  target: false,                                 // no planned length
  pen: (colors: (red, green, rgb("#FF99DD"))),   // your own pen colours
))
```

What is not named is on: a deck that says nothing gets the whole view. `tools: false` takes the drawing bar away.

A tile that is switched off takes its keys with it: with `clock: false`, `t` and `⇧t` do nothing and no longer stand in the key bar.

The colours are Typst colours, not strings, and there may be more or fewer than four. `c` steps through them in turn.

### 9.3.2 Light or dark

The view follows the system setting of the machine it stands on (`prefers-color-scheme`), and `l` contradicts it when the room is not what the operating system thinks. The choice holds for the session and survives a reload.

It is expressly **not** the deck's palette: the lectern is a tool and should read the same whatever the room does.

> **TIPP**
> The preview shows the next step, not the next slide: what the next keypress does, be it a new slide or one more reveal on the current one. The label above it says which.

### 9.3.3 Drawing

You draw on the running slide in the speaker view and the strokes appear on the projected one: the presenter has a trackpad in front of them, and the canvas is across the room.

Strokes stick to their slide, so paging away and back brings them with you. `x` clears the current slide, `z` takes back the last stroke, `c` changes colour.

### 9.3.4 A clock the class can see

`t` asks for a number of minutes, and the wall then carries nothing but a clock: black ground, white digits, `m:ss`, large enough to read from the back row. It replaces the slide rather than sitting on it – the twin of `b`, only with something on it. It is meant for the break, the group work, the experiment being set up.

| Key | What it does |
|---|---|
| `t` | ask for minutes; `Enter` starts it, `Esc` leaves it |
| `t` (while it runs) | end the clock, the slide is back |
| `⇧→`, `⇧←` | one minute more or less, while it runs as well |
| `→` (or any other paging key) | ends it and uncovers the slide |

In the speaker view it has a tile of its own, `class clock`, beside the tile `target (min)`. The two do not look alike: the target duration is a field of whole minutes you set once per talk, the class clock a running `m:ss` with a bar that empties. While none runs a dash stands there, and while no talk window answers a longer one.

At zero it does not stop but carries on to `+0:01` in the deck's accent colour, with the word „over" above it. Nothing blinks and nothing chimes. The overtime is capped at the duration itself and at thirty minutes. At the lectern the whole tile turns over at the same moment, in the warning colour, so that the teacher sees the overtime no later than the class does.

> **ACHTUNG**
> `t` when nothing else is on the wall. No clock while you are talking: a clock running beside a sentence pulls the eye for the whole talk. It is therefore not laid over the slide but replaces it, and whoever goes on talking presses it away.

> **HINWEIS**
> Like black and freeze, the clock lifts on its own if the speaker window goes away; the talk window has no key against it. Reload the talk window and the clock comes back – further along, not from the start.

### 9.3.5 The pointer

`m` switches the pointer between the pen and the embedded frame. In pointer mode the pen rests and a press on an embedded frame lands in the talk window instead. Press, drag and release travel as fractions of the stage, so a small laptop window and a large canvas hit the same point of the document.

Where the embedded document can mirror itself, as a GeoGebra applet does, the live one in front of you is operated instead and the projected copy follows.

> **ACHTUNG**
> It reaches listeners, not the browser's own widgets. A checkbox toggles and a button fires, because a click carries its activation behaviour along; an `input type=range` does not move, because a browser only drags its own slider for input it trusts. Whoever builds for this listens rather than relying on a native control.

### 9.3.6 Blacking out and freezing

`b` blacks the room out, `e` freezes the projected image while you page ahead in private. Steering works from either window, and either one may be reloaded: they find each other again, and the strokes come back.

> **ACHTUNG**
> Both lift by themselves shortly after the speaker window is closed. If that window stays open but no longer carries a deck, a one-minute deadline applies instead – and a stalling talk window on top of that can put it off indefinitely. That is the one known corner in which the room stays dark.

The speaker view opens on one keypress, and a **real** keypress is the condition: `window.open` without a user gesture falls to the popup blocker.

## 9.4 Less motion

Someone who has turned on „reduce motion" in their operating system gets a quieter deck. The runtime asks for `prefers-reduced-motion: reduce` afresh on every step and every frame, so switching it on in the middle of a talk takes effect at the next keypress. There is nothing to configure.

The setting says „less motion", not „no motion": **opacity stays, travel goes.** An entrance still says „this is new", but nothing crosses the slide any more.

| What | What becomes of it |
|---|---|
| Entrances | Every effect keeps its opacity and loses its travel: `fade-up`, `fade-down`, `fade-left`, `fade-right`, `scale`, `scale-down`, `rise` and `blur` become a plain cross-fade. `fade` and `none` are left as they are. `duration` and `delay` do not change. |
| `enter: "draw"` | The pen holds still, the fade remains. The drawing **is** the travel, and what is left when it is taken out is exactly the cross-fade that ran underneath it anyway. |
| Slide transitions | Every kind but `none` becomes the cross-fade, over the same `transition-duration`. `none` stays the hard cut. |
| Magic move | Does not happen. Nothing flies, and the slide changes the way it would change without a morph. |
| Flip book | Stands still on one frame. Without `loop` and without `pingpong` that is the last one, where it would have come to rest anyway; only the way there falls away. With `loop` or `pingpong` it is |

| What | What becomes of it |
|---|---|
| | frame zero. `still` does not apply: the frame for paper is typeset content and is not in the HTML at all, which carries only the frames themselves. |
| `scene` | Jumps from stop to stop. The frames in between still sit in the file, but none of them is shown. What falls away is exactly the travel; the stops themselves are not travel, they are the content. |
| `after: "dimmed"` | Stays. A point stepping back changes its opacity and does not move. |
| The progress bar in the speaker view | Jumps to its new width instead of gliding there. |

Two things are deliberately left alone.

**Video.** A video is content, not decoration. Whoever does not want it to start by itself writes `autoplay: false`.

**Embedded documents.** The runtime does not reach into a foreign document. The setting does: `matchMedia("(prefers-reduced-motion: reduce)").matches` is true inside the frame as well, so anyone animating something there writes their own `@media` rule.

> **HINWEIS**
> No switch lets a deck overrule the setting. Where a motion really carries the argument, it belongs in words as well – and those are read by the people who never see it run.

# 10 Three outputs from one source

The talk for the canvas, the deck to read afterwards, and the handout to write on, without a second version to keep in step.

## 10.1 The slide deck

The PDF run without further arguments gives one page per slide, in the size of the canvas. Every element that moves in the browser stands in its final state: what is revealed is there, and where several versions share one place, the last one. Notes, transitions and bridge jobs produce no output and fall away by themselves.

## 10.2 The handout

One argument turns the deck into a handout on A4:

```
#show: presentation.with(handout: 3)   // three slides per page
```

`handout` takes `true` (two per page) or a number from 1 to 6 and applies only to the PDF. The slides are not typeset again, only made smaller, so a handout cannot differ from what stood on the canvas.

Beside or below each slide stands its note; where a slide has none, ruled lines take its place. Up to two slides per page the notes stand **below** and the slide takes the full width; from three on they stand **beside**.

## 10.3 What the paper leaves out — and what to plan for

| On the slide | On paper |
|---|---|
| `anim`, `stagger`, `#pause` | everything visible, in the same place and the same room |
| `alternatives` | the last version only, in the shared box |
| `morph` | the content of each slide – the chain becomes the calculation |
| `embed`, `geogebra` | `fallback`, otherwise a placeholder with `label`; `link` below it |
| `video` | the `poster`, otherwise a grey panel |
| `flipbook` | a single picture: `still` or `render(0.0)` |
| `scene` | a single picture: `still` or the last stop |
| `speaker-note` | beside its slide in the handout, nothing in the plain slide deck |
| `transition`, `bridge-job` | nothing – they belong to the motion alone |

> **TIPP**
> Whoever knows a handout will come sets `fallback` and `link` while writing the slide. Afterwards every embedded place has to be visited a second time.

## 10.4 All three in one run

Since Typst 0.15 one compilation can write several files. `bundle` writes talk, deck and handout at once:

```
#bundle(
  theme: themes.lesson,
  title: [Completing the Square],
```

```
  handout: "handout.pdf",
)[
  = A section
  == A slide
  Text.
]
```

```
typst compile --features bundle,html --format bundle talk.typ out
```

`html`, `slides` and `handout` are file names, `none` leaves that output out, and `per-sheet` is the number of slides on a handout page. Everything else goes to `presentation` unchanged. The counters start afresh per output: the deck numbers 1, 2, 3 and does not carry on where the HTML version stopped.

> **ACHTUNG**
> The bundle is experimental on Typst's side and needs `--features bundle,html`. A file that uses `bundle` compiles **only** with `--format bundle`; a plain `typst compile talk.typ talk.pdf` stops with „constructing a document is only supported in the bundle target". To keep both routes open, put the body in a `#let` and call `presentation` by hand.

## 10.5 Notes

`speaker-note` files a note with the slide. It stands in the body or as the argument `note` on `slide`:

```
== The Pythagorean Theorem
#speaker-note[Show the dissection first, then the formula.]
```

The note appears in the speaker view and on the handout. It produces nothing in the deck PDF.

A note has to carry text: the speaker view transports it as a string and the handout prints it where there is text. A note built purely out of layout – a `fit`, a bare `rect`, an image – is refused with a message. What is meant to be **seen** belongs on the slide.

## 10.6 Two clocks for the class

`t` starts the **full-screen clock**. It covers the slide edge to edge, with digits the back row can read: the room is on a break. Paging ends it and uncovers the slide again.

`⇧T` starts the **pinned clock**. It stands *on* the slide and leaves the task underneath in place, so paging deliberately does not end it. In the presenter view it can be dragged with the mouse, and it travels along in the talk window.

Both ask for the minutes first and only then run. `⇧←` and `⇧→` give a minute more or less; the same key again ends the clock.

What a deck knows about the pinned clock it writes with `class-clock`:

```
#slide[
  = Group work
  #class-clock(12)
  Find three examples in pairs.
]
```

Nothing starts from that: `⇧T` offers the twelve minutes and the speaker confirms or changes them. The deck knows how long the task was meant to take, the room decides how long it gets.

# 11 Making it your own

The aim: a deck that looks like yours and not like the package.

## 11.1 Choosing a theme

Five ship with the package, made for different occasions rather than one slide in five colours. The title sits in a bar, free, or under a line; the progress indicator grows, or is missing.

| Theme | Made for |
|---|---|
| `themes.default` | A conference talk. Title in a coloured band, bar of progress. |
| `themes.lesson` | A lesson. White paper, a running head, tinted panels with the caption inside, no progress bar. |
| `themes.night` | A darkened room. Dark ground, one signal colour. |
| `themes.plain` | Getting out of the way. White, black, one grey. |
| `themes.editorial` | Reading rather than presenting. A serif face, a quiet rule. |

## 11.2 Changing one

A theme is a plain dictionary, so `+` is all it takes:

```
#show: presentation.with(theme: themes.lesson + (accent: blue))
```

`theme(...)` builds one from scratch. Its eight colour entries are the same eight a palette carries, listed in the next section. Four keys take one word each:

**`header`** — `"band"`, `"plain"` or `"run"`
**`footer`** — `"fraction"`, `"number"`, `"center"` or `"none"`
**`progress`** — `"bar"`, `"top"`, `"tick"` or `"none"`
**`box`** — `"bar"` or `"label"`

A typo in one of those four is an error, not a silent default: the message names the values it accepts. `title-slide` and `section` are functions instead – those two are whole pictures, not variations on one theme. The typographic keys and the measures are in the API reference.

## 11.3 Colour, separately: palettes

A theme says how a slide is **built**; a **palette** says what colour it is. The two vary separately, which is why they are separate arguments. A palette overwrites **partially**, only the entries written down:

```
#show: presentation.with(theme: themes.lesson, palette: (accent: blue))
#show: presentation.with(theme: themes.lesson, palette: palettes.dark)
```

The eight entries are exactly a theme's colour entries: `paper` the ground of the slide, `ink` the body text, `strong` the carrying dark colour, `accent` the signal colour, `muted` the secondary matter, `surface` the ground of a card, `border` its edge, and `inverted`, whether light text stands on a dark ground. An entry that does not exist is refused: `palette: (acent: blue)` stops with a message.

Five ship with the package, and each composes with each of the five themes:

| Palette | Where it comes from |
|---|---|
| `palettes.light` | The colours of `themes.default`, so this one changes nothing. |
| `palettes.mono` | The greys of `themes.plain`, two moved so it passes the contract below. |
| `palettes.textbook` | The colours of `themes.lesson`, one grey moved. |
| `palettes.parchment` | The laid paper of `themes.editorial`, two tones moved. |
| `palettes.dark` | The dark ground of `themes.night`, with a deeper accent. |

That is why the dark room needs no theme of its own: **darkness is a palette rather than a design.** `themes.lesson` under `palettes.dark` is still the lesson design, only dark.

`themes.night` stays a theme all the same. Its cyan glows on night's own ground but all but vanishes on the ground an inverted slide lays behind it, so `palettes.dark` takes a deeper blue that holds on both while the theme keeps the cyan it was designed around.

> **ACHTUNG**
> Two colours of a theme are not palette entries: `title-fill` and `rule-fill`. Whether they follow is up to the theme. All five bundled ones let them follow – either as a function of the palette, `title-fill: p => p.strong`, or as `none`, which means the accent. A theme of your own that names a fixed colour there keeps it under every palette: a colour someone named out loud is not swapped behind their back.

## 11.4 The colours of a theme

The five bundled themes fill those eight roles differently:

```
#import "@preview/typstage:0.1.0": themes
#themes.night.accent      // the theme's signal colour, as a colour
```


`card` and `callout` take their colours from the running theme, so a change of theme recolours them. Where one card is to look different, it takes `color:` and `fill:`.

> **TIPP**
> A colour that carries meaning — blue for the function, orange for its slope — is best fixed once at the top of the file and handed on where it belongs: `card(color: …)`, `callout(color: …)`, `ggb-style(color: …)`.

Independently of the theme the package hands out four colour constants — `dark`, `accent`, `paper` and `muted` — the default look. They are handy where a slide needs a shade and the theme is not being changed; whoever swaps the theme is better served by its entries.

## 11.5 Inverting one slide

For the slide that carries a single number there is `invert`. The ground becomes the palette's text colour and the text becomes its ground; `muted`, `border` and `surface` are mixed from those two, `strong` and `accent` carry over unchanged. Running head, footer, slide number, progress bar, `card` and `callout` follow.

In the heading notation it is a marker in the slide body, like `#pause`:

```
== Reached by 2026
#invert
#statement[74 %]
```

In the argument notation it is an argument of `slide`:

```
#slide([Reached by 2026], invert: true)[#statement[74 %]]
```

> **ACHTUNG**
> Only a regular slide inverts. Title and section slides are whole pictures the theme draws itself, and neither takes the argument.
>
> The `#invert` marker is found wherever the body can be walked: nested in a block, an align, a table cell or a grid, in the slide's own heading, and behind `#set` and `#show` rules. It is **not** found where the content is handed to a closure. Measured, that is nine: `context`, `fit`, `anim`, `card`, `callout`, `tiles`, `cue`, `stagger` and `alternatives`. There the slide is left as it is, without a word. Where you need one of those, write `slide(invert: true)`, which never depends on the walk.

## 11.6 The contrast contract

The bundled palettes are measured before they ship, against the WCAG 2 contrast ratio. Seven pairs are checked:

| Pair | At least | What for |
|---|---|---|
| `ink` on `paper` | 4.5 | body text on the slide |
| `ink` on `surface` | 4.5 | body text in a card |
| `muted` on `paper` | 4.5 | footer, subtitle, running head |
| `accent` on `paper` | 3.0 | rules, progress bar, marker |
| `accent` on `ink` | 3.0 | the same on an inverted slide |
| `accent` on black | 3.0 | the overtime of the full-screen clock |
| `border` on `paper` | 1.2 | hairlines |

The second to last has no palette role as its ground: the full-screen clock is black from edge to edge whatever the deck's palette says, and its overtime digits are set in the accent.

All five bundled palettes are checked automatically, upright and inverted. A colour moved there that breaks the contract stops the build and names the number it missed.

> **ACHTUNG**

**The contract holds only the bundled palettes.** A palette of your own faces no gate: it is neither warned about nor recoloured. `palette-report(…)` hands the same measurement back as a list:

```
#for f in palette-report((paper: white, ink: black, surface: white,
                          muted: luma(55%), accent: blue, border: luma(86%))) [
  #f.pair: #calc.round(f.ratio, digits: 2) (wants #f.min) #f.ok \
]
```

`contrast(a, b)` is the arithmetic itself and takes any two colours.

**And the five themes do not all pass it.** They were measured before the palettes existed, and the result stands here rather than being quietly coloured away:

| Theme | What falls short |
|---|---|
| `themes.default` | nothing, all seven pairs hold |
| `themes.lesson` | `muted` on `paper` |
| `themes.night` | `accent` on `ink` |
| `themes.plain` | `muted` on `paper`, and `accent` on `ink` |
| `themes.editorial` | `muted` on `paper`, and `accent` on `paper` |

None of those colours was changed: moving them would have changed every deck already written, and what `muted` carries is secondary matter – slide number, subtitle, running head. Anyone who wants the numbers met lays the matching palette over the theme:

```
#show: presentation.with(theme: themes.editorial, palette: palettes.parchment)
```

ACHTUNG
**The text colour is never inferred from the fill.** A muted sage such as `#aebdb3` reads as „light“ to a luminance rule, yet white on it measures 1.96 to 1, far under the 4.5 that body text wants. So the package measures with `contrast` and recolours nothing on its own.

The one exception lives in the theme, not the palette. Where a theme uses `strong` as **text** – the heading in `themes.lesson`, the section title in `themes.plain` – it picks between `strong` and `ink` by contrast against the ground, because one colour cannot serve as both a dark band and text on a dark ground.

## 11.7 The canvas

`width`, `height` and `margin` on `presentation` set the canvas. The default is 16:9 on an A4 width; 4:3 is `width: 800pt, height: 600pt`. Everything the theme draws scales along.

## 11.8 Typography

`style` is a show rule applied to the slides **and** to the moving parts:

```
#show: presentation.with(
  style: it => { set text(font: "Libertinus Serif"); set par(justify: false); it },
)
```

ACHTUNG
A tracked element is typeset a second time in a frame of its own, and that frame never sees a `#set` rule from the document. Shared typography therefore has to go here: a `#set text` after the show rule reaches the slides but not the flying pieces, and the difference only shows up mid-flight.

> For the shapes typstage draws itself there is a second route, label rules before `#show: presentation`. They reach more, including the header, the footer and the title slide. See /Labels: reaching every shape the package builds/ below.

## 11.9 Building blocks for the body

Six functions for the body of a slide: `card`, `callout`, `side-by-side`, `tiles`, `statement` and `fit`. The coloured ones take the running theme's colours unless told otherwise.

### 11.9.1 card: the named box

```
#card(title: [Power function])[$f(x) = x^n$ with $n in NN$.]
```

**POWER FUNCTION**

$f(x) = x^n$ with $n \in \mathbb{N}$.

`number:` puts a numbered disc in front, for the running order where the number belongs to the matter – `card(number: 2, title: [Second step])[…]`. `color:` tints the bar, `fill:` the panel.

### 11.9.2 callout: the one that has to stick

```
#callout[The exponent decides the symmetry.]
```

**MERKE**

The exponent decides the symmetry.

`title:` changes the caption (it follows the document language by default), `color:` its colour, and `title: none` leaves it off.

### 11.9.3 side-by-side: two columns

The usual case for a slide with something to look at: the drawing or the applet on the left, the words on the right.

```
#side-by-side(
  card(title: [Even exponent])[Symmetric about the $y$ axis.],
  stagger[
    - $f(-x) = f(x)$
    - Range $W = [0; oo[$
  ],
)
```

**EVEN EXPONENT**

Symmetric about the $y$ axis.

- $f(-x) = f(x)$
- Range $W = [0; \infty[$

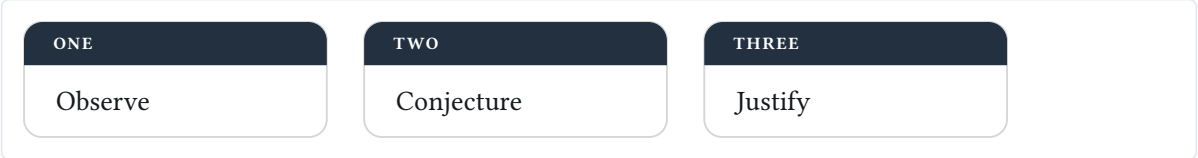
`split:` takes the column widths; the default gives the first column a little more, because that is usually where the picture goes. More than two columns are allowed — they then share the width equally, unless `split:` names as many values.

`equal: true` makes every column the height of the tallest; without it each box stands as tall as its own text, and two cards side by side look differently weighted. The row is measured once and its height handed on as a length, which is why `equal` reaches `card` and `callout` rather than arbitrary content.

### 11.9.4 tiles: the grid that numbers its own reveals

Each tile appears one step after the one before, without an `anim` per tile and a number counted up.

```
#tiles(
  card(title: [one])[Observe],
  card(title: [two])[Conjecture],
  card(title: [three])[Justify],
)
```




`columns:` sets how many there are (up to three by default). `stride: 0` puts them all on the same step and staggers only through `stagger`, in milliseconds — a wave runs through the grid then, instead of a sequence of keypresses:

```
#tiles(stride: 0, stagger: 90, [A], [B], [C], [D])
```

`duration:` and `easing:` are those of `anim` and apply to every tile alike: a grid moves as one thing. Given neither, the presentation's duration and the house curve apply.

### 11.9.5 statement: the large claim

```
#statement[$ a^2 + b^2 = c^2 $]
```

$$a^2 + b^2 = c^2$$

`statement` asks for the full width explicitly and centres within it. A bare `align(center, …)` inside a tracked element cannot: the element is only as wide as its content.

### 11.9.6 fit: working content into the room it has

For the one piece whose size is not written in the deck: the wide table out of the analysis, the generated chart, the list from a data file. Left alone, such a block runs over the edge of the slide – visibly in the PDF, cut away in the browser, where the slide sits in a frame of fixed size.

```
== Regression results
#fit(wrap: false, my-table)
```

`fit` measures the block against the place it stands in and scales it geometrically, so the proportions are kept and no factor is given by hand. The result is the same in the HTML and in the PDF.

**Width first, then smaller.** The block is offered the full width before it is measured. A paragraph or a list then wraps into the space instead of shrinking, and only what is still too tall afterwards is scaled. A block that already fits is left untouched.

**`wrap: false` for anything that lays itself out in columns.** A table, a chart or a drawing does not wrap when it is offered a narrower width, it rearranges itself – under the default `wrap: true` the columns are squeezed, the digits overlap, and nothing is scaled at all. `wrap: false` measures such a block exactly as it stands. It is the one setting worth knowing before the first use.

**It only shrinks.** `grow: true` also blows up what is smaller than its place, for the one large number meant to fill the slide. `shrink: false` takes the shrinking away and leaves only the growing.

```
#fit(grow: true)[42%]
```

`width` and `height` take `auto`, a length or a ratio. On `height: auto` the block takes what is left over below the rest of the slide, so a fit under two bullet points reckons with them. That backfires inside a `card`: the box becomes slide-tall, is cut off at the bottom, and whatever follows falls off the slide. Give `height:` explicitly inside a card.

> ACHTUNG
> **No reveal inside a `fit`.** Two things do not survive being measured. A `pause` is found by walking the slide body, and a fitted block is a closure that walk cannot enter, so its steps fall away silently. And a measured block gets no bounded height to reckon against, so a tracked element inside one cannot reserve the room for its marker.
>
> `fit` therefore stops with a message that names the thing, for `pause`, `anim`, `stagger`, `alternatives`, `morph`, `tiles`, `video`, `embed`, `flipbook`, `build`, `scene`, `camera` and `cue` – in both outputs, and also when the fit sits inside another fit. Put the fit **inside** the reveal rather than around it:
>
> ```
> #anim(fit(wrap: false, my-table))   // yes
> #fit(anim(my-table))                // no
> ```

`speaker-note` and `bridge-job` are allowed inside a `fit`. The other direction is not: a note made only of a `fit` carries no text, and `speaker-note` refuses it with a message.

The arithmetic is taken from mosaic, which took it from Touying 0.7.4; Touying credits the work on it to Andreas Kröpelin (Polylux PR 91) and to ntjess.

### 11.9.7 overflow: the checking pass before the talk

`fit` answers the one block whose size you already suspect. `overflow` answers the question you cannot ask slide by slide: does anything in this deck run over the room it has? It measures every slide body and names the ones that do not fit.

```
#show: presentation.with(overflow: "error")
```

It is off by default and meant to be switched on for a run, not left on while writing. A build script can raise it from the command line instead of editing the deck, which is how the seventeen example decks of this package are measured on every push:

```
typst compile --features html --format html \
  --input typstage-overflow=error deck.typ deck.html
```

The input raises, it never lowers: of the two settings the stricter one wins, `"none"` < `"record"` < `"error"`, so no run can quietly switch a check off in passing.

A deck needs this more than a document does: an overrun on a page stands past the margin where the eye catches it, while a slide goes into a frame of fixed size and what sticks out is cut away.

`"none"` — nothing is measured. The default.

`"error"` — the whole deck is built, and it then stops with **every** place at once rather than with the first. One run, the whole list.

`"record"` — it carries on and files a queryable record per finding instead, for a tool or a build script. Typst gives a package no warning channel, so `"record"` prints nothing by itself.

The message names the slide, the step and the amount (shortened here):

```
error: assertion failed: typstage: 2 slides run over the room the body has. …
  slide 2, from step 1 at the earliest: 311.14pt too tall, 675.76pt of content in 364.61pt of
room
  slide 3, from step 2 at the earliest: 296.49pt too tall, 661.1pt of content in 364.61pt of
room
Shorten the slide, split it, or put the block that does not fit into fit(). …
```

**Why the step says „at the earliest“.** A slide is the same height on every step – only what is **drawn** changes, not the room reserved for it. The step is therefore a lower bound, exact only where the thing that overruns is itself a reveal. **The slide is named correctly either way**, and that is the part to act on. On paper no step is named, because every step stands on the page at once; in the records that shows as `step: 0`.

The records are read with `typst eval`, and for that the deck has to be on `overflow: "record"` – on `"error"` this command stops with the error instead:

```
typst eval --target html --features html --in deck.typ \
  'query(<typstage-overflow>).map(e => e.value)'
```

which gives one entry per finding:

```
[{"slide":2,"step":1,"height":675.76,"room":364.61,"over":311.14},
 {"slide":3,"step":2,"height":661.1,"room":364.61,"over":296.49}]
```

> **HINWEIS**
> **What the check does not see.** Only the height is measured, so a body that is too wide goes unnoticed; `fit` is the answer to that case. A `height: 100%` in the body measures 0 and a `1fr` collapses. Anything drawing outside its own layout box – `scale`, `move`, `place` with an offset – is invisible to a measurement. Title and section slides are never measured: they have no body block to overrun.
>
> And one thing is reported where nothing shows: trailing spacing, a `v()` at the end of a body, takes room in the measurement and draws nothing.

In HTML the pass costs up to half again as long per deck; on paper it costs next to nothing.

### 11.9.8 drift: the check for scenes that travel

`overflow` asks whether a slide fits its room. `drift` asks the other question one cannot check slide by slide: does a scene stand still while the talk pages through it?

A drawing is as large as what it holds, a CeTZ canvas above all. Change the content across the stops of a `scene` and every frame comes out a different size, so the drawing sits somewhere else in its box each time: paging moves the whole picture although only one point was meant to move. Every scene measures its frames, and `drift` says what happens with the findings.

`"error"` — the whole deck is built, and it then stops with **every** scene at once. The default.
`"record"` — it carries on and files a queryable record per finding.
`"none"` — nothing is measured at all.

```
#show: presentation.with(drift: "record")
```

The message names the slide, the step and the numbers (shortened here):

```
error: assertion failed: typstage: 1 scene draws frames of different sizes. …
  slide 4, from step 1: 28 frames in 19 different sizes, up to 28.35pt apart across and
53.86pt down
```

The records are read exactly as the overflow ones, with `query(<typstage-drift>)`, and for that the deck has to be on `drift: "record"`.

**Why this check is on where `overflow` is not.** Only decks that use `scene` pay for it, while `overflow` measures every body of every deck. And what this one finds is invisible while writing: every frame on its own looks right, and only paging shows the drawing travelling. Only the browser branch measures; on paper a single still image stands there, and a still image does not travel.

> **HINWEIS**
> It flags the scene; it does not fix it. A drawing that sets itself to `100%` measures the same on every frame and drops out of the check, rightly so: it already has a fixed frame. And a drawing that only grows to the right and downwards is reported although its ink does not move – `steady: false` on that scene takes it out of the check.

### 11.9.9 Slides without a title

A bare `==` leaves the title band off; the body starts at the top and gets the height the band would have taken. This is the slide for the one large formula, and the target of a morph that is to fly into the middle:

```
==
#place(center + horizon, morph(<derivative>, text(size: 2.4em)[
  $f'(x) = lim_(h -> 0) (f(x+h) - f(x)) / h$
]))
```

In the argument form all three spellings are allowed: `slide[body]` without a title, `slide(none)[body]` explicitly without, `slide([Title])[body]` with.

## 11.10 Labels: reaching every shape the package builds

Every shape typstage draws itself carries a fixed Typst label: the ground, the header band, the slide title, the footer, the progress indicator, the card, the callout, the statement, the title and section slides, the box that stands in for a video. An ordinary `show` rule reaches it – no theme key, no fork.

```
#import "@preview/typstage:0.1.0": *

#show label("ts-slide-header-band"): set rect(fill: rgb("#4c1d95"))
#show label("ts-slide-title"): set text(fill: rgb("#fde047"), style: "italic")
#show label("ts-card"): set block(fill: rgb("#eef2ff"))
#show label("ts-statement"): set text(fill: rgb("#be123c"), weight: "bold")

#show: presentation.with(theme: themes.default)
```

Two kinds of rule cover all of it. The **surfaces** – grounds, bands, hairlines, bars, boxes – take `set rect(..)`, `set block(..)`, `set circle(..)` or `set line(..)`. The **type** takes `set text(..)`. Both apply identically in HTML and PDF, with one exception: the six labels under /Media and handout/ are drawn only in the PDF, because the browser puts the real `<video>` or `<iframe>` in their place.

> **ACHTUNG**
> **For the surfaces** the short form works and the long one does not:
>
> ```
> #show label("ts-slide-progress"): set rect(fill: green)          // yes
> #show label("ts-slide-progress"): it => { set rect(fill: green); it }   // no
> ```
>
> The short form puts the style rule **around** the element it matched, the long one puts it **inside** – and inside the rectangle there is no second rectangle for it to reach.

> For the 16 type labels the two spellings are equivalent: what sits inside the matched element there is the text, and a rule reaches that from within.

### 11.10.1 Where the rule has to stand

**Before** `#show: presentation`. That one place reaches everything: the slide background, the chrome layer with header, footer and progress, the title slide and every moving piece.

The `style` hook does **not**. It is wrapped around the slide **body**, and header, footer, progress and the two whole-picture slides are built beside it. Measured, all 38 rules one by one: from `style` exactly the 13 that stand in the body take effect – `ts-card…`, `ts-callout…`, `ts-statement` and the three `ts-media-…` surfaces. The other 25 stay silent, without a warning.

> ACHTUNG
> A `show` rule written **after** `#show: presentation` does not reach a tracked element (`anim`, `morph`), for the reason given under Typography: in the browser every moving piece is typeset a second time in a frame of its own, and that frame never sees a `#show` rule from the document body.
>
> ```
> #show: presentation.with(theme: themes.default)
> #show label("ts-statement"): set text(fill: green)   // too late
> == A slide
> #statement[still]
> #anim(statement[moving])
> ```
>
> Here `still` comes out green and `moving` black. With the same rule one line further up, both look alike. The PDF does not show the difference, because nothing is typeset twice there.
>
> This holds for every `#show` rule, not only for label rules.

### 11.10.2 What a label rule changes and what it does not

Reachable is whatever the package does **not** set explicitly: for type everything, for surfaces `fill`, `stroke` and `radius`.

`width` stands as an argument everywhere and is therefore nowhere reachable. `height` has three exceptions: `ts-card`, `ts-card-bar` and `ts-callout` get their height as `auto`, and `auto` cannot beat a rule.

```
#show label("ts-card"): set block(height: 150pt)   // works
#show label("ts-card"): set block(width: 30%)      // does not
```

The first line blows the card up to 150 pt and pushes the callout under it off the slide. On the chrome surfaces and the handout frame neither line does anything; what a `width` rule seems to change there are the blocks **inside** the content, see the next box.

The slide's **arrangement** is not reachable either. How tall the header builds, how far the rule sits under the title, where the bar goes – no `show` rule reaches into that. The theme keys are there for it: `head-gap`, `band-height`, `rule-size` and the rest.

> ACHTUNG
> A rule on `block` or `rect` reaches **inwards**: it holds for the labelled surface and for every block inside it. For `fill`, `stroke` and `radius` that is caught – the card puts the document's own setting back inside. For the spacings it is not, and then a label rule moves the slide:
>
> ```
> #show label("ts-card"): set block(below: 60pt)
> == A slide
> ```

```
#card(title: [Card])[Body]
#callout(title: [Note])[Remember this]
```

The callout then moves down, and everything below it with it – by the spacing given minus the block spacing already there, **per edge**. Setting `above` and `below` at once gives twice the shift.

That is not a promise but a side effect of Typst's style rules. Labels are meant for type and surface; for spacings, use the building blocks' own arguments or the theme keys.

### 11.10.3 The complete inventory

What stands here exists; what exists stands here. The names follow one scheme: `ts-`, then the **place**, then the **part**. Places are `slide` (the ordinary slide), `title-slide`, `section-slide`, `card`, `callout`, `statement`, `media` and `handout`.

The mnemonic: `slide` in **front** means the ordinary slide; `slide` behind `title` or `section` means that kind of slide. So `ts-slide-title` is the title of an ordinary slide and `ts-title-slide-title` the title of the title slide. Reaching for the wrong one of such a pair does nothing at all, silently.

A label the current theme does not draw – a header band under `header: "run"`, say – is not on that slide, and a rule on it does nothing.

**The ordinary slide**

| Label | What it is | Rule |
|---|---|---|
| `ts-slide-ground` | The slide's ground | `rect` |
| `ts-slide-header-band` | The header band, only under `header: "band"` | `rect` |
| `ts-slide-header-text` | The running header of number and section, only under `header: "run"` | `text` |
| `ts-slide-header-rule` | The hairline under it, only under `header: "run"` | `rect` |
| `ts-slide-title` | The slide title, under all three header styles | `text` |
| `ts-slide-title-rule` | The rule under the title, only when `rule-size > 0pt` | `rect` |
| `ts-slide-footer` | The footer line | `text` |
| `ts-slide-number` | The slide number in it | `text` |
| `ts-slide-footer-rule` | The hairline above it, only when `footer-rule > 0pt` | `rect` |
| `ts-slide-progress` | The progress bar, or under `progress: "tick"` the marker that travels | `rect` |
| `ts-slide-progress-track` | The track it travels along, only under `progress: "tick"` | `rect` |

**The title slide**

| Label | What it is | Rule |
|---|---|---|
| `ts-title-slide-ground` | Its ground | `rect` |
| `ts-title-slide-band` | The band along the top edge, only in `themes.lesson` | `rect` |
| `ts-title-slide-title` | Its title | `text` |
| `ts-title-slide-subtitle` | Its subtitle | `text` |
| `ts-title-slide-rule` | The accent stroke; `themes.editorial` has two, `themes.plain` none | `rect` |
| `ts-title-slide-byline` | The line of author and date | `text` |

### The section slide

| Label | What it is | Rule |
|---|---|---|
| `ts-section-slide-ground` | Its ground | `rect` |
| `ts-section-slide-bar` | The bar along the left edge, only in `themes.lesson` | `rect` |
| `ts-section-slide-title` | Its title | `text` |
| `ts-section-slide-rule` | The accent stroke; `themes.night` has two, `themes.lesson` none | `rect` |
| `ts-section-slide-parent` | The line above it naming the sections this one hangs under. Only from the second structure level on, so never at `slide-level: 2` | `text` |

A section slide has no subtitle in typstage, so the list names none.

### The building blocks in the body

| Label | What it is | Rule |
|---|---|---|
| `ts-card` | The card: surface, border, rounding and all of its contents | `block` |
| `ts-card-bar` | The coloured tab above it, only under `box: "bar"` | `block` |
| `ts-card-title` | Its caption | `text` |
| `ts-card-disc` | The disc of the number, only with `number:` | `circle` |
| `ts-card-number` | The numeral in it | `text` |
| `ts-card-body` | Its body | `text` |
| `ts-callout` | The callout: surface, bar, rounding. The bar on the left is not a label of its own, it is this one's left `stroke` – `set block(stroke: (left: 4pt + red))` recolours it | `block` |
| `ts-callout-title` | Its caption | `text` |
| `ts-callout-body` | Its body | `text` |
| `ts-statement` | The large statement. `size` acts as a factor on it, because `statement` measures in `em` | `text` |

### Media and handout

| Label | What it is | Rule |
|---|---|---|
| `ts-media-fallback` | The box that stands in for a moving element in the PDF. A container only, so a `radius` rule on it is not visible while a `fill` rule is | `block` |
| `ts-media-fallback-empty` | The grey box inside it when no `fallback:` was given. That one has a surface | `block` |
| `ts-media-poster` | The grey area of a `video` without a `poster:` | `rect` |
| `ts-handout-frame` | The framed box of one slide on the handout page | `block` |
| `ts-handout-lines` | The writing lines beside or below it | `line` |
| `ts-handout-note` | The speaker note, where there is one | `text` |

> **HINWEIS**
> **A theme with its own title slide draws none of these labels.** `title-slide` and `section` in a theme are functions and paint their picture themselves, so whoever brings their own loses the labels of that

> slide kind, and nothing warns about it. Which of the bundled themes draws what stands in the /What it is/ column.
>
> **The invisible markers carry none.** Every moving element paints an invisible marker rectangle around itself, and `pin` does the same for a single glyph – machinery, not a shape, so neither carries a label.
>
> **Typst labels and the runtime's CSS classes are two separate namespaces.** `.ts-slide` in the stylesheet is a slide's `<section>` in the browser, `ts-slide-title` is a Typst label – one hyphen apart and unrelated. Typst's HTML export does put a `data-typst-label` attribute on some shapes and not on others. That is Typst's own by-product, not a promise of this package: do not build CSS on it.

## 11.11 `info()`: what the deck knows about itself

Labels say how a shape looks, not what stands in it: the slide number, the fraction, the chapter in the running header. `info()` hands those out:

```
#context {
  let deck = info()
  [#deck.section.title #h(1fr) #deck.slide.number / #deck.slide.total]
}
```

Every number the package prints on a slide comes out of this dictionary, so a hand-built footer and the built-in one cannot disagree. What comes back:

| Field | What is in it |
|---|---|
| `title`, `subtitle` | The deck's title and subtitle, as `presentation` or a `title-slide` received them |
| `author`, `date` | From the same place. `date` is whatever was passed, a `datetime` or content |
| `slide.number` | This slide. Counted the way the footer counts, so title and section slides are not in it |
| `slide.total` | How many slides are counted |
| `slide.numbered` | Whether this slide is one of them. `false` on a title and on a section slide |
| `step.number` | The step the calling content itself stands on |
| `step.total` | How many steps this slide has |
| `section.number` | Which section is running, `0` before the first |
| `section.total` | How many sections the deck has |
| `section.title` | Its title, or `none` before the first |
| `levels` | One entry per structure level, outermost first. Empty at `slide-level: 1` |
| `outline` | The whole structure, one entry per section slide in the order they come |

`section` always means the level directly above the slide. At the default `slide-level: 2` that is the only level there is, and `section` is then `levels.last()` without its `depth`. A deck with more than one level – see „More than two levels" – finds them in `levels` and in `outline`:

| Field of an entry | What is in it |
|---|---|
| `levels.at(i).depth` | The heading level, `1` for `=`, `2` for `==` |
| `levels.at(i).title` | The title, or `none` while no section of that level is running |

| Field of an entry | What is in it |
|---|---|
| `levels.at(i).number` | Which section of that level it is in the **whole** deck. It never goes back, so it also reads as progress |
| `levels.at(i).total` | How many sections that level has in the whole deck |
| `levels.at(i).index` | Which one it is **under the same parent**, what Beamer prints as `1.2` |
| `levels.at(i).count` | How many siblings it has there. `index` and `count` are `0` while no section of that level is running |
| `outline.at(j).depth` | The same for an entry of the outline |
| `outline.at(j).title` | Its title |
| `outline.at(j).number` | The same count as `levels.at(..).number`. Comparing the two says whether the entry is past, running or still to come |
| `outline.at(j).here` | Whether the slide being shown is that very entry. Only a section slide can be, and only a theme's own `section` function can read it there: a section slide has no body for a deck to write into |

A progressive agenda therefore needs no second count:

```
#context {
  let d = info()
  stack(spacing: 0.6em, ..d.outline.map(e => {
    let running = d.levels.at(e.depth - 1).number
    text(
      weight: if e.number == running { "bold" } else { "regular" },
      fill: if e.number <= running { black } else { luma(60%) },
      [#h((e.depth - 1) * 1.4em)#e.title],
    )
  }))
}
```

One number stands apart: the speaker view and the overview count **every** slide, title and section slides included, while `info().slide.total` counts the way the footer counts and leaves those out.

### 11.11.1 Two counts, not one

A slide is one picture, a step is one press of the arrow key. The deck counts both, and this manual keeps the two words apart.

`step.number` is the step the calling content itself stands on: `1` in the body of a slide, and inside an `anim`, a `stagger` or an `alternatives` the step of that reveal – the first of them where the reveal covers several. So a display naming the current step has to sit inside the reveals; the browser typesets nothing anew:

```
#let where = context {
  let d = info()
  [Step #d.step.number of #d.step.total]
}

== Four versions
#alternatives(where, where, where, where)
```

Paging through, that prints „Step 1 of 4“ up to „Step 4 of 4“.

On paper there is no current step: the page shows the slide in its final state, everything at once, and `step.number` equals `step.total`.

> **HINWEIS**
> `step.total` counts what the runtime in the browser counts – for every building block that consumes a step, and the PDF names the same number.

### 11.11.2 Where a hand-built footer goes

typstage draws no footer on a title or a section slide, and nothing belongs in the counter slot there. `slide.numbered` says when that is the case:

```
#let footline = context {
  let d = info()
  let number = if d.slide.numbered [#d.slide.number / #d.slide.total] else []
  place(bottom + right, text(size: 12pt, fill: muted, number))
}
```

On an ordinary slide it goes into the body:

```
== A slide
#footline
The text of the slide.
```

On the title and the section slides it has to go into the theme: both are functions, and a function wrapped around another adds to it instead of replacing it.

```
#let base = themes.default
#let with-foot(f) = (t, s, geo) => { f(t, s, geo); footline }

#show: presentation.with(
  theme: base + (title-slide: with-foot(base.title-slide),
                 section: with-foot(base.section)),
)
```

> **ACHTUNG**
> **Not through `style:`.** `style: it => { footline; it }` looks like the shortcut that puts the footer on every slide at once. But `style` is also the template each moving element is typeset with a second time, so whatever **draws** in there is drawn again inside every sprite: a deck with three reveals per slide showed the footer four times over. In the body it is drawn once.
>
> `style:` is for typography – typeface, size, colour, leading – and for that it is exactly right: background and sprite need the same.

> **HINWEIS**
> A footer placed in the body sits at the bottom of the **body**, not at the bottom of the slide; the theme's `foot-gap` lies in between. A `dy:` on the `place` moves it where it belongs.

> **ACHTUNG**
> `info()` reads the state of the slide being typeset and therefore needs a `context` around it. **Before** the presentation there is nothing to read, and it stops with a message rather than handing out zeros. **After** it there is: whoever passes the slides as arguments and writes an `info()` below the call still gets the last slide's numbers. In the show-rule notation nothing comes after the deck anyway.

### 11.11.3 `deck-outline()`: how the deck is cut

`info()` says **where** you stand, not how the whole thing is divided. A navigation bar needs exactly that: which slides belong to which section. `deck-outline()` hands it over, one entry per section, in the order they come:

```
#context for a in deck-outline() [
  - #a.number. #a.title -- slides #a.first to #a.last (#a.count)
]
```

Each entry carries `depth`, `number`, `title`, `first`, `last` and `count`. `first`, `last` and `count` are **transitive**: a depth-1 section counts the slides of its sub-sections too, so a bar does not show a zero for every top-level heading. A section with nothing under it has `none` for `first` and `last`, and `0` for `count`.

Only headings standing **between** slides count. A heading **inside** a slide, `slide(none)[= Every map lies]`, is a slide title and opens no section; a deck written exclusively that way gets an empty list back. So put the `=` between the slides, not into them; `examples/gliedern.typ` shows how.

> **HINWEIS**
> It reads only what every slide already carries – no `query`, no second walk over the document, the same answer in both outputs.

> **ACHTUNG**
> A foreign package looking for the structure through `query(heading)` finds nothing: the heading notation splits the body at its headings and copies `depth` and `body` out, dropping the element itself. That holds in **both** outputs. `deck-outline()` is the answer to it.

# 12 Handing it on

Getting the talk to where it will be given.

## 12.1 One file

`assets: "inline"` is the default and writes the runtime into the HTML. The result is one file that runs from a memory stick, from a download folder, from an email attachment. No server, no network, nothing loaded afterwards. The runtime adds about 330 KB to every deck.

## 12.2 Beside the file

`assets: "split"` refers to two files next to the HTML instead, and `runtime-files` gives you their names and contents so you can write them out:

```
#for f in runtime-files {
  // f.name and f.content
}
```

That is worth it where many decks are published together: the browser caches the runtime once, and every deck after the first pays nothing for it. On short decks that is about half of what visitors load.

`assets: (cdn: "https://…")` points at a directory on a server or a CDN. It takes a dictionary, not a bare string: a string falls through unread, and the page then links names that are not there.

## 12.3 Hosting

The HTML file is static. Anything that serves files serves it: GitHub Pages, a university web space, an S3 bucket.

Media travels beside the file. `video("clip.mp4")` refers to a file that has to lie next to the HTML; without it an uploaded deck shows an empty frame where it worked locally.

A deck opened from `file://` behaves like one from a server, speaker view included.

# 13 What it cannot do

The limits, in one place, so they are not discovered in front of an audience.

## 13.1 Accessibility

The hardest limit, and it follows from the design decision on the first page. The slides are SVG outlines: the letters in them are drawn as paths, not as text. Nothing is selectable, nothing is searchable, and a screen reader finds nothing to read – no text alternative, no reading order.

What does work: the document carries a `lang` attribute from `text.lang`, navigation is fully operable from the keyboard, and colour and contrast are the theme's and therefore yours to set. A viewer whose system asks for less motion gets less: `prefers-reduced-motion` is read at run time. For the same everywhere, set `transition: "none"` and `enter: "none"`.

> **ACHTUNG**
> If someone in the room reads with a screen reader, hand out the PDF as well and say what is on each slide. The PDF from the same source carries real text.

## 13.2 A tracked element with no area

A tracked element takes its place in the browser from a rectangle Typst paints around it. Content with no area leaves that rectangle with none either, and a viewport of zero scales everything inside it to nothing: the element is in the page and cannot be seen. On paper it stands.

The two usual cases – a vertical rule and a `place` – are handled by the package itself.

```
#anim(at: 2, place(top + left, dx: 20pt, dy: 50pt,
                   rect(width: 20pt, height: 20pt)))
```

The rest is reported, not lost: a marker with no width, one with no height, and one nested deeper than four tracked elements each print once to the browser's console.

```
typstage: the tracked element 3 on slide 4 has a marker with no width.
Its sprite is given a viewport of that extent, and a viewport of zero
scales everything inside it to nothing: the element is in the page, with
its path and its colour, and cannot be seen. On paper it stands. Put it
in a box with a size, or give the element a width.
```

The check can only run in the browser. Whether content has an area is not a question the document can answer; only there is the rectangle measurable.

## 13.3 Reach

Tested in Chrome, Firefox and Safari on macOS, and on an iPhone. Not tested: older browsers, Windows, Android. The runtime uses the Web Animations API, `ResizeObserver`, `PointerEvent` and CSS `zoom`, so a browser from before about 2023 is likely to fall short somewhere.

## 13.4 Size and speed

A slide is typeset once per state, and every tracked element once more, in a frame of its own. Compile time therefore grows with steps, not with slides, and `flipbook` grows with frames.

The example decks compile in seconds and stay under 5 MB. A hundred slides with a flip book on each is a different matter – measure it rather than guess.

# 14 When nothing happens

The traps, in roughly the order they are usually hit.

**No HTML export** — `--features html` is missing. The export is experimental in Typst, not in this package.

**The deck is empty but for the title** — the two notations have been mixed. Either write `= …` and `== …`, or hand `slide(...)` calls to `presentation`. A `slide(...)` inside the body of a show rule makes no slide and no error either.

**The first paragraph is missing** — content before the first heading belongs to no slide. Text there stops the compile — see „Text that belongs to no slide“. An image there goes without a word.

**The slide titles ignore a `#set heading`** — the `#set` comes after the show rule, so the titles are already outside its reach. `style:` reaches them.

**`#pause` does nothing** — it sits in a grid cell or a table, and there is no body there to split. `anim` goes anywhere content goes.

**A transition or an entrance is refused by name** — the package does not know it. The message names every effect there is.

**The bullets beside an applet start at step three** — `embed` uses no step, but something before it did. Count the reveals, not the elements.

**A flying equation has the wrong font** — a tracked element is typeset in a frame of its own, which `#set` does not reach. `style:` on `presentation` does.

**An embedded frame stays empty and gets no jobs** — the document has not announced itself with `postMessage({typstage: 1, ready: 1})`.

**An applet frame stays empty** — the applet comes from `geogebra.org`. Without a network, point `codebase` at a local copy.

**A `ggb-run` command has no effect** — it is one of GeoGebra’s scripting commands, which `evalCommand` does not accept. Use `ggb-set`, `ggb-style`, `ggb-show` or `ggb-hide` instead.

**The build stops naming two applets** — two frames on one slide and no `target`. Nothing is guessed.

**The applet’s colours change after paging back** — GeoGebra takes the next colour of its palette on a rebuild. Fix the colour on `"1-"`.

**A circle in the applet is an ellipse** — the x range and the y range of `ggb-view` do not match the shape of the box.

**A tween does not play** — it sits on step 1, where tweens are set to their target instead of played, or it was given a range instead of a step number.

**Two sliders lie on top of one another** — for a slider made with `Slider`, `position` counts in pixels, not in coordinates.

**A point cannot be dragged in the speaker view** — it was made with `Point(k, 0.3)` and is pinned to that parameter.

**An embedded frame is right on the laptop and tiny on the projector** — its content is sized in `px` instead of `em`. Inside a zoomed frame one CSS pixel is one point of the slide.

**„constructing a document is only supported in the bundle target“** — the file uses `bundle` and therefore needs `--format bundle`.

**The speaker view does not open** — `window.open` needs a real keypress; a script cannot stand in for it.

# 15 API reference

Generated from the comments in the source files: presentation and slides first, then the building blocks, then media and the bridge, and last the measurements and colours.

## 15.1 The presentation

### 15.1.1 `bundle`

```
bundle(
  body,
  html: "talk.html",
  slides: "slides.pdf",
  handout: none,
  per-sheet: 3,
  ..args,
)
```

All outputs in one run.

Since 0.15 Typst can write several files from one compilation. That fits this package, since everything sits in one source anyway: the talk, the slide deck and the handout differ only in target and in one setting. Instead of compiling three times, once:

```
typst compile --features bundle,html --format bundle talk.typ output
```

```
#bundle(
  theme: themes.lesson,
  title: [Completing the Square],
  handout: "handout.pdf",
)[
  = A section
  == A slide
  Text.
]
```

`html`, `slides` and `handout` are file names; `none` leaves out that output. `per-sheet` is the number of slides per handout page. Everything else goes to `presentation` unchanged.

Two things worth knowing. The bundle is explicitly experimental in Typst. And a file that uses `bundle` can **only** be compiled with `--format bundle`: `typst compile talk.typ talk.pdf` aborts with „constructing a document is only supported in the bundle target“. Anyone who wants both writes the body into a `#let` and calls `presentation` by hand.

Verified: the counters start over for each output. The slide deck numbers its slides 1, 2, 3 and does not continue counting where the HTML version left off, even though Typst runs introspection across the whole bundle.

| Name | Vorgabe |
|---|---|
| `body` | – |
| `html` | `"talk.html"` |
| `slides` | `"slides.pdf"` |
| `handout` | `none` |

| Name | Vorgabe |
|---|---|
| `per-sheet` | `3` |
| `..args` | – |

### 15.1.2 `presentation`

```
presentation(
  ..slides,
  title: none,
  subtitle: [],
  author: [],
  date: none,
  assets: "inline",
  theme: themes.default,
  palette: (:),
  transition: "slide",
  speaker-view: (:),
  transition-duration: 420,
  duration: 520,
  style: it => it,
  width: auto,
  height: auto,
  margin: auto,
  handout: false,
  overflow: "none",
  drift: "error",
  slide-level: 2,
)
```

Build the deck.

Two notations, the same output. Either the slides as arguments:

```
#presentation(title-slide(title: [Title]), section[Part], slide([First])[…])
```

… or as a show rule, and then the headings separate the slides:

```
#show: presentation.with(title: [Title], transition: "slide")
= A section
== A slide
Content …
```

Two targets, one source:

```
typst compile deck.typ deck.html --format html --features html
typst compile deck.typ deck.pdf
```

`slide-level:` is where the deck is cut. A heading **above** it becomes a section slide, a heading at it or below it becomes a slide. The default is 2, and that is the rule this package always had: `=` opens a section, `==` a slide.

```
#show: presentation.with(title: [Analysis], slide-level: 3)
= Part I
== Sequences
=== What a sequence is
A map from the naturals.
```

`= Part I` and `== Sequences` each become a section slide, `===` becomes the slide. Both transition slides come for free: a section heading **is** the transition slide here, so there is nothing to switch on and no hook to write. `slide-level: 1` makes every heading a slide and leaves the deck without any structure level.

A deeper level is drawn more quietly by all five bundled themes, smaller and with the titles it hangs under set above it. A theme of its own reads `s.depth` and `s.parents` off the section record and may ignore both, and then every level looks alike.

What the deck knows about its structure is in `info()`: `section` is unchanged and always means the level directly above the slide, `levels` has one entry per structure level, and `outline` is the whole thing.

`theme:` determines the whole look: colors, typeface, title bar, footer, progress, title and section slide. Bundled are `themes.default` (the default), `themes.lesson`, `themes.night`, `themes.plain` and `themes.editorial`; each of them can be varied with `themes.night + (accent: blue)`. The `style` hook stays untouched by this and sits further **inside**: whatever is set there overrides the theme.

`palette:` changes the colors and leaves the design alone. It is a dictionary over the eight color entries and it overwrites **partially**, so `palette: (accent: blue)` moves the accent and nothing else. Five are bundled, `palettes.light`, `palettes.mono`, `palettes.textbook`, `palettes.parchment` and `palettes.dark`, and each of them composes with each theme:

```
#show: presentation.with(theme: themes.lesson, palette: palettes.dark)
```

Two colors of a theme are not palette entries: `title-fill` and `rule-fill`. All five bundled themes let them follow, either as a function of the palette or as `none`, which means the accent and follows with it. A theme of your own that names a fixed color there keeps it under every palette, which is deliberate.

Both changed type with this: reading `themes.X.title-fill` used to give a color and now gives a function, and `rule-fill` gives `none` where it gave the accent. Writing them, `themes.X + (title-fill: red)`, is unchanged.

`speaker-view` says what the presenter view shows. Everything is on unless switched off, so a deck that says nothing gets the whole thing:

```
#show: presentation.with(speaker-view: (
  clock: false,                       // no class clock
  target: false,                      // no planned length
  pen: (colors: (red, green, blue)),  // the drawing bar's colours
))
```

A tile that is switched off takes its keys with it: with `clock: false`, `t` and `⇧t` do nothing and no longer stand in the key bar. A view that advertises a key which does nothing is worse than one that is missing it. `tools: false` removes the drawing bar the same way.

The PDF is a handout: one page per slide, every tracked element in its final state. What belongs only to the motion, the notes, the slide transitions, the bridge jobs, are state updates without output and fall away by themselves.

`overflow` is a checking pass over the deck, off by default. It measures every slide body against the room the theme gives it and names the ones that do not fit, with the earliest step on which the overrun can be on the screen. Title and section slides are not measured: the theme draws them with `place` and they have no body block.

- `"none"`: nothing is measured. The default.
- `"error"`: the whole deck is built, and it then stops with **every** place at once rather than the first.
- `"record"`: it carries on and files a record per finding instead, for a tool to read. The deck has to be on `"record"` for this; on `"error"` the command below stops with the error too:

```
typst eval --target html --features html --in deck.typ \
  'query(<typstage-overflow>).map(e => e.value)'
```

The same setting can be raised from the command line, so a build script can measure a deck without editing it:

```
typst compile --features html --format html \
  --input typstage-overflow=error deck.typ deck.html
```

The input raises, it never lowers. Of the two the stricter one wins, `"none"` < `"record"` < `"error"`, so a run cannot switch off a check the deck asked for.

It is not meant to stay on while writing. Measured over the six example decks: in HTML it costs noticeably more time, between 1.2 and 1.5 times depending on the deck and on how the process start is accounted for. On paper it costs a few milliseconds per deck, small but repeatable: there the check runs without the step arithmetic.

Why a deck of slides needs this more than a document does: a slide goes into an SVG frame of fixed size and is scaled in the browser, so what sticks out is cut away or drawn beside the slide. A page one leafs through shows an overrun; a talk one clicks through shows it at the projector.

`drift` is the second check, and unlike `overflow` it is on. Every `scene` measures its frames, and a scene whose frames come out different sizes is named: a CeTZ canvas is as large as what it holds, so a wider frame puts the drawing somewhere else inside its box and paging through it the whole picture travels while only one point should move.

- `"error"`, the default: the deck is built and then stops with every scene at once. `scene(steady: false)` says the frames of that one scene are meant to differ and takes it out of the check.
- `"record"`: it carries on and leaves a record per finding, for a tool to read, the same way `overflow: "record"` does:

```
typst eval --target html --features html --in deck.typ \
  'query(<typstage-drift>).map(e => e.value)'
```

- `"none"`: the frames are not measured at all.

On by default where `overflow` is not, and for two reasons. Only decks that use `scene` pay for it at all – measured on a scene of 28 CeTZ frames, 434 ms without and 536 ms with, so about 100 ms for that scene – where `overflow` measures every slide of every deck and costs 1.2 to 1.5 times the whole compilation. And what it finds is invisible while writing: every frame on its own looks right, and only paging through shows the drawing travelling. Only the browser branch measures. On paper a scene is one still image, and a still image does not travel.

| Name | Vorgabe |
|---|---|
| `..slides` | – |
| `title` | `none` |
| `subtitle` | `[]` |
| `author` | `[]` |
| `date` | `none` |
| `assets` | `"inline"` |
| `theme` | `themes.default` |
| `palette` | `(:)` |
| `transition` | `"slide"` |
| `speaker-view` | `(:)` |

| Name | Vorgabe |
|---|---|
| `transition-duration` | `420` |
| `duration` | `520` |
| `style` | `it => it` |
| `width` | `auto` |
| `height` | `auto` |
| `margin` | `auto` |
| `handout` | `false` |
| `overflow` | `"none"` |
| `drift` | `"error"` |
| `slide-level` | `2` |

## 15.2 Slides

### 15.2.1 `class-clock`

```
class-clock(minutes)
```

Plans a class clock for this slide: how many minutes the work on it is meant to take.

It starts nothing. `Shift+T` in the presenter view offers the number, the speaker confirms or changes it, and only then does the clock run – the deck knows how long the task was meant to take, the room decides how long it actually gets. A slide carries at most one; a second call replaces the first, like a second `speaker-note`.

### 15.2.2 `deck-outline`

```
deck-outline()
```

How the deck is cut, section by section.

`info()` says where you are; this says how the whole thing is divided. One entry per section, in the order they come, each with the slides beneath it:

```
#context for a in deck-outline() {
  [#a.number. #a.title -- slides #a.first to #a.last (#a.count)]
}
```

`first`, `last` and `count` are transitive: a depth-1 section counts the slides of its sub-sections too. A section with no slides under it has `none` for `first` and `last`, and 0 for `count`.

Read straight off the state every slide already carries. No `query`, no second walk over the document, and the same answer in both outputs.

### 15.2.3 `info`

```
info()
```

What the deck knows about itself, read from inside a slide.

```
#context {
  let deck = info()
  [#deck.section.title #h(1fr) #deck.slide.number/#deck.slide.total]
}
```

It is the same reading the built-in chrome does. Every number the package prints on a slide, the footer, the fraction, the length of the progress bar and the running header, comes out of this function and out of no second count, so a hand-built footer and the built-in one cannot disagree.

What comes back, as a dictionary:

- `title`, `subtitle`, `author`, `date`: the deck's own particulars, as `presentation` or a `title-slide` received them.
- `slide.number`, `slide.total`: this slide and how many there are. Counted the way the footer counts, so title and section slides are not in it.
- `slide.numbered`: whether this slide is one of the counted ones. It is `false` on a title slide and on a section slide, and `number` then holds the last slide counted before it, 0 on a cover that opens the deck. A footer can therefore leave its counter slot clear instead of printing a zero into it.
- `step.number`, `step.total`: this deck counts in steps as well as in slides, which no footer can guess at. `number` is the step the calling content itself stands on: 1 in the body of a slide, and inside an `anim`, a `stagger` or an `alternatives` the step of that reveal, its first one where it covers several. On paper a slide is one page in its final state, so `number` is `total` there.
- `section.number`, `section.total`, `section.title`: which section the slide belongs to, how many the deck has, and its title. The section is always the level directly above the slide, so at the default `slide-level: 2` this is the `=` heading and nothing about it has changed. Before the first such heading, `number` is `0` and `title` is `none`. A deck at `slide-level: 1` has no structure level at all, and then all three read `0`, `0` and `none`.
- `levels`: one entry per structure level, from the outermost inwards, so `levels.last()` is the same section as above and `levels.first()` the outermost part. Empty at `slide-level: 1`. Each entry carries `depth` (1 for `=`, 2 for `==`), `title` (`none` while no section of that level is running), `number` and `total` (counted across the whole deck, the way `section` counts), and `index` and `count` (counted among the siblings under the same parent, which is what Beamer prints as `1.2`). `number` never goes back, so it also reads as progress; `title`, `index` and `count` clear when the level above them moves on.
- `outline`: the whole structure of the deck, one entry per section slide in the order they come, each with `depth`, `title`, `number` (the same count as in `levels`) and `here`, which is `true` only on that section slide itself. Comparing an entry's `number` with `levels.at(entry.depth - 1).number` says whether it is past, running or still to come, and that is how a progressive agenda is built.

Only in a context. **Before** any presentation has run there is nothing to read and this stops with a message rather than handing out zeros. **After** one it does not: whoever passes the slides as arguments and writes an `info()` below the call still gets the last slide's numbers. Clearing the deck's own record at the end would close that, and it was measured: a slide carrying one reveal beside a `tiles` went from no layout warning to three „did not converge" ones. A corner nobody stands in is not worth that, and in the show-rule notation nothing comes after the deck anyway.

### 15.2.4 `section`

```
section(title, depth: 1, transition: none)
```

A section slide.

`depth` is the level in the heading hierarchy the section stands on: `1` for `=`, `2` for `==` and so on, up to one below the deck's `slide-level`. In the heading notation it comes from the heading; here it is written out. The five bundled themes draw a deeper section more quietly, and a theme of your own reads it from `s.depth`.

| Name | Vorgabe |
|---|---|
| `title` | – |

| Name | Vorgabe |
|---|---|
| `depth` | `1` |
| `transition` | `none` |

### 15.2.5 `slide`

```
slide(
  title: none,
  note: none,
  transition: none,
  invert: false,
  ..rest,
)
```

A regular slide.

`title: none`, or a bare `==` in heading form, leaves out the title bar; the body then gets the whole area.

`invert: true` sets this one slide in the palette turned around, for the slide that carries a single number. The ground becomes the palette's text color and the text becomes its ground; `muted`, `border` and `surface` are mixed from those two; `strong` and `accent` carry over unchanged. The chrome follows, so the running header, the footer and the progress bar are set in the same colors as the slide under them.

Only a regular slide inverts. A title slide and a section slide are whole pictures the theme draws itself, and three of the five bundled themes build them from colors an inversion would not reach; neither takes the argument.

| Name | Vorgabe |
|---|---|
| `title` | `none` |
| `note` | `none` |
| `transition` | `none` |
| `invert` | `false` |
| `..rest` | – |

### 15.2.6 `speaker-note`

```
speaker-note(body)
```

A note for the presenter view, which `n` opens in a second window.

The note has to carry text: the presenter view transports it as a string, so a note made purely of layout would arrive nowhere. That is refused with a message rather than silently dropped.

### 15.2.7 `title-slide`

```
title-slide(title: [], subtitle: [], author: [], date: none)
```

The title slide.

| Name | Vorgabe |
|---|---|
| `title` | `[]` |
| `subtitle` | `[]` |
| `author` | `[]` |
| `date` | `none` |

### 15.2.8 `transition`

```
transition(kind, ..spec)
```

How this slide comes in, otherwise the presentation's setting applies.

- `"none"`: hard cut.
- `"fade"`: cross-fade, nothing moves.
- `"slide"`, `"push"`, `"cover"`, `"uncover"`: sliding; `from` says where the new slide comes from: `"right"` (default), `"left"`, `"top"`, `"bottom"`. `push` shoves the old one out, `cover` lies down on top, `uncover` pulls the old one away.
- `"zoom"`: `direction: "in"` (default) grows the new one towards you, `"out"` lets it step back from the front.
- `"blur"`: blurred across.
- `"iris"`, `"wipe"`: an aperture. `direction: "open"` (default) opens the new slide out, `"close"` shuts the old one over it. For the wipe, `from` additionally says which edge it starts at.
- `"flip"`, `"cube"`: rotation in space; `axis: "y"` (default) turns about the vertical, `"x"` about the horizontal.

Backwards each one runs as a true reversal. If a morph meets the slide it cross-fades regardless: the movement is then carried by the morph.

Under `prefers-reduced-motion: reduce` every kind but `"none"` becomes the cross-fade, over the same duration. See the manual.

### 15.2.9 `invert`

The same thing in the heading notation, written into the slide body.

```
== Reached in 2026
#invert
#statement[74 %]
```

A marker, like `#pause`, because a heading carries no arguments. Unlike `#pause` it is only looked for, never split on, so the walk goes all the way down: it is found in a `block`, an `align`, a table cell, a grid, however deeply nested, in the heading itself, and behind `#set` and `#show`. It is not found where the content is handed to a closure – in `context`, `fit`, `anim`, `card` or `alternatives` – and there nothing happens and nothing is said. Measured, those five are the whole of it; `slide(invert: true)` is the form that never depends on the walk.

It prints nothing, so it may stand anywhere in the body; it inverts the whole slide either way, not the part after it.

## 15.3 Revealing, moving, staggering

### 15.3.1 `alternatives`

```
alternatives(
  ..variants,
  start: auto,
  align: top + left,
  enter: "fade",
  duration: auto,
  easing: auto,
  inline: false,
  morph: false,
)
```

Several versions of the same thing, each replacing the one before.

```
#alternatives(
  $ (a + b)^2 $,
  $ (a + b)(a + b) $,
  $ a^2 + 2 a b + b^2 $,
)
```

`inline: true` keeps the whole thing in the running line, for versions of a single word or formula.

They all stand in the same place, in a box as large as the largest of them, so nothing around them jumps as they change. Each takes one step; the last one stays for the rest of the slide.

`morph: true` lets the versions fly into one another instead of replacing one another. They all stand in the same place, so the flight is no distance at all and what you see is the glyphs rearranging themselves where they stand – which is what a rewritten formula does:

```
#alternatives(morph: true,
  $ (a + b)^2 $,
  $ (a + b)(a + b) $,
  $ a^2 + 2 a b + b^2 $,
)
```

It works because the versions carry one name and step ranges that do not overlap, and a morph flies from step to step as readily as from slide to slide. A name of your own instead of `true` is allowed and is only needed where the flight has to continue onto the next slide.

A morph has no entrance and no easing curve, so `enter:` and `easing:` are refused rather than quietly dropped. `duration:` is read and is the time of the flight.

On paper only the last one is set, in the same box, so the page keeps the spacing of the slide. Printing all of them would pile them on top of one another.

| Name | Vorgabe |
|---|---|
| `..variants` | – |
| `start` | `auto` |
| `align` | `top + left` |
| `enter` | `"fade"` |
| `duration` | `auto` |
| `easing` | `auto` |
| `inline` | `false` |
| `morph` | `false` |

### 15.3.2 `anim`

```
anim(
  body,
  at: auto,
  enter: "fade-up",
  exit: "fade",
  after: "hidden",
  duration: auto,
  delay: 0,
  easing: auto,
)
```

Reveal content on particular steps.

`at` is a step selector. `auto`, the default, takes the next free step, so consecutive `anim`s reveal one after another without any numbering. Otherwise: `2` (from step two on), `"1-2"`, `(2, 4)`, `"3"`. An explicit number also moves the cursor along, so a following `auto` carries on after it instead of starting over.

`enter` applies in both directions: paging back plays the same effect in reverse, taking the entrance back. `exit` only concerns a real departure, when an element falls out of its range while moving forward.

`after` says what the element does once its range is behind it, and it has two values.

- `"hidden"`, the default and what an `anim` has always done: it goes, playing `exit`, and keeps the room it had.
- `"dimmed"`: it stays and is drawn muted, so a point remains legible after the talk has moved on. Nothing moves and nothing is recoloured; the element settles to 65 percent opacity, and paging back brings it up again. That number is measured, and the manual says against what.

On paper `after` does nothing at all. A page shows every step at once, and a point that is only quiet because the talk has moved past it has no „past“ on a handout. This is the same rule that already holds for `"hidden"`: what leaves its range in the browser is still printed.

`after` needs a range that ends. `at: auto` and `at: 3` run to the end of the slide, and an element that never leaves has no after; the package says so instead of doing nothing. `at: "3"` is that one step, `at: "2-3"` a range.

Under `prefers-reduced-motion: reduce` every effect keeps its opacity and loses its travel, so `enter` and `exit` become a plain cross-fade of the same length. `after: "dimmed"` is unaffected: it changes opacity and nothing else. See the manual.

`easing` is the curve the element moves on – for the entrance, the departure and the dimming alike. `auto` is the package’s own curve; `"out-back"` overshoots and swings back, `"linear"` arrives at an even pace. A name that does not exist is an error at compile time and not a silent default.

| Name | Vorgabe |
|---|---|
| `body` | – |
| `at` | `auto` |
| `enter` | `"fade-up"` |
| `exit` | `"fade"` |
| `after` | `"hidden"` |
| `duration` | `auto` |
| `delay` | `0` |
| `easing` | `auto` |

### 15.3.3 `build`

```
build(
  draw,
  steps: 2,
  start: auto,
  enter: "fade",
  duration: auto,
  easing: auto,
)
```

A drawing or a diagram that comes into being step by step.

A CeTZ canvas and a lilaq diagram are one piece, not many: Typst hands out the finished setting, and what was a line and what was a data series in it cannot be reached from outside any more. So there is no `anim` around a part of a drawing. What there is, is the drawing itself, as often as one wants it.

`draw` is called once per step and is handed a question. The examples call it `from`, because it says exactly what `at:` says elsewhere; the name is the deck's own, since it is the lambda's parameter.

- `from(k, value)` gives `value` back once the k-th piece is due, and otherwise the same thing made of air: a colour with alpha 0, a stroke with a transparent brush, a text in `hide`. The piece is therefore never really missing, and every stage measures the same to the point.
- `from(k)` says the same as a boolean, for everything that cannot be recoloured. In CeTZ that is where `hide(…, bounds: true)` belongs.

Whatever carries no number stands there from the start.

```
#build(from => cetz.canvas({
  import cetz.draw: *
  line((0,0), (4,0))                          // there from the start
  line((4,0), (4,3), stroke: from(2, black))  // from step 2
  content((2,3.4), from(3, [hypotenuse]))     // from step 3
}), steps: 3)
```

`steps` is the number of stages and hence the number of steps the drawing takes on the slide. It is said and not guessed: what `draw` does with its question is nobody's business from outside.

`start` is `auto`: the drawing begins on the next free step and pushes the cursor along by `steps`, the way `stagger` and `alternatives` do. A number sets the first step itself.

Exactly one stage is drawn at a time, and that is not a saving but the only arrangement that yields the picture that would stand there if the drawing were set once. Ink adds up: three layers of the same lilaq diagram against one, and 3.7 percent of the pixels differ by more than 8 of 255, the largest deviation 99 – axes, labels and the half-transparent box of the legend get painted three times and grow fatter by it.

On paper only the last stage is set, in a block of the same size: a page shows every step at once, and stacked stages would be overprint. The cursor still runs there, so that `info().step.total` names the same number in both outputs.

Under `prefers-reduced-motion: reduce` nothing changes: the stages fade, they do not travel, and what would fall away is a motion that is not there.

| Name | Vorgabe |
|---|---|
| `draw` | – |
| `steps` | `2` |
| `start` | `auto` |
| `enter` | `"fade"` |

| Name | Vorgabe |
|---|---|
| `duration` | `auto` |
| `easing` | `auto` |

### 15.3.4 `camera`

```
camera(
  target,
  at: auto,
  margin: 16pt,
  duration: 700,
  easing: auto,
)
```

Move in on one detail of the slide, and back out again.

```
#pin(<detail>, card[The measuring head])
…
#camera(<detail>)
```

The camera aims at a `pin`, and at nothing else. That is the package's word for a named piece of a slide, its marker is exactly the rectangle the runtime already measures, and it sits wherever content sits: around a card, around a cell of a table, around one subterm of a formula. Nothing has to be given in coordinates, and nothing has to be counted.

`at` is a step selector, as everywhere else, and the slide is seen through the camera for exactly as long as it is active. That answers the way back out with the notation that is already there: `at: "3"` moves in on step three and back out on step four, `at: "3-5"` holds the crop across three steps, `at: 3` keeps it to the end of the slide.

`auto`, the default, is the next free step **closed**: in on it, out on the one after – and never step one, because step one is the slide as it is entered, and a camera there would mean nobody ever saw the slide whole. That differs from `anim`, where `auto` runs to the end of the slide, and it differs on purpose. An entrance has no natural end – what has appeared stays. A camera move has one: one always comes back out, and coming back out is a keypress like any other, so it is counted like one and shows up in `info().step.total`.

`margin` is how much of the slide stays around the detail, measured in the unzoomed slide. The camera fits the detail plus that margin into the frame; the smaller of the two directions decides, so the whole of it is seen.

A detail that is already as large as the slide gives nothing to travel to, and then the slide stays whole.

Two pins of the same name on one slide are framed together, and the camera shows the box around both.

If two camera moves are active on the same step, the later one in the source wins.

On paper there is no camera. The slide is set whole, exactly as it would be without one – but the steps are counted there too, so the handout's footer names the same number as the talk.

Under `prefers-reduced-motion: reduce` the camera jumps to the crop instead of travelling to it. The package's rule everywhere else too: what stays is the destination, what goes is the travel.

| Name | Vorgabe |
|---|---|
| `target` | – |
| `at` | `auto` |
| `margin` | `16pt` |
| `duration` | `700` |

| Name | Vorgabe |
|---|---|
| `easing` | `auto` |

### 15.3.5 `cue`

```
cue(name, ..items, start: auto, spacing: 0.65em)
```

Reveal one after another: a list or several blocks.

Two notations, the same function:

```
#stagger[
  - this first
  - then this
]
#stagger(card[left], card[right])
```

For a list, the bullet marks are set here rather than left to `list`: only this way does the mark belong to the tracked element. If it stayed with the list, it would sit in the background and be there before its point appears.

`start` is `auto`: the sequence continues where the slide left off. `stride: 0` makes everything appear on the same step and staggers only through `stagger`, in milliseconds.

`dim: true` turns the sequence into a walk: the point being discussed stands there, the ones before it stay legible but muted. Every point then holds exactly its own step instead of the rest of the slide, and rests at `anim`'s `after: "dimmed"` from the next step on. Paging back brings each one up again.

Two things follow from that and are worth knowing before reaching for it. The last point dims too as soon as the slide has a further step after it, because then the walk has moved on from it as well. And `stride: 0`, which puts every point on one step, makes them all dim together on the next. A group that is revealed in whatever order it is called out.

For points that have no order of their own: what the class names gets shown, in the order it comes rather than the order it stands in. The digits `1` to `9` choose; the speaker view shows which digit belongs to which point.

```
#cue("ablesen", start: 2)[
  - positive und negative Werte
  - tiefster und höchster Wert
  - Abnahme und Zunahme
]
```

The group owns as many steps as it has points, and the order changes nothing about that. Everything that hangs on the step count – the progress bar, `info().step.total`, the overflow check, the handout – is therefore untouched.

Set, the list keeps its reading order: a point not yet named holds its place, so nothing jumps when it arrives later.

| Name | Vorgabe |
|---|---|
| `name` | – |
| `..items` | – |
| `start` | `auto` |
| `spacing` | `0.65em` |

### 15.3.6 `cue-layer`

```
cue-layer(name, number, body, enter: "fade")
```

Something that appears together with one point of an adaptive group.

A drawing layer, a picture, a sentence beside it: it shares the step with its point and therefore travels with it, without having to be linked.

```
#cue-layer("ablesen", 1, schicht-vorzeichen)
```

The group has to stand **before** its layers in the source, because a layer looks up which step its point was given. Standing after them, the package says so rather than quietly doing nothing.

| Name | Vorgabe |
|---|---|
| `name` | – |
| `number` | – |
| `body` | – |
| `enter` | `"fade"` |

### 15.3.7 `morph`

```
morph(
  name,
  body,
  at: "1-",
  duration: 900,
  match: "auto",
  inline: true,
)
```

Magic move: the same `name` twice, and the thing flies across.

Twice on two adjacent slides, or twice on one slide with `at:` selectors that do not overlap. The runtime pairs the flights step by step as well as slide by slide.

The name is a string or a label: `morph(<pythagoras>, …)`.

`duration` is 900 ms, not the duration of the presentation: a flight across the slide takes more time than a simple fade-in, and in real decks the default was overridden in 161 of 165 cases. `auto` falls back to the presentation's duration.

`match` is `"auto"`, `"glyph"` (always per glyph) or `"block"` (always as one rectangle).

Two names may be equal on the **target** slide. The runtime looks the source up by name but iterates over the targets, so two targets sharing a name both start from the same place and the glyph visibly splits in two. `at` is almost always right as it is. A morph is present from the first step on: a flight target must already be there when the slide is entered. Because paging back swaps the roles, that holds for both ends.

Delaying is only worthwhile for the **first** link in a chain, for example when the formula should appear together with its tile. It is allowed exactly when the preceding slide carries no morph of the same name; the package checks this at compile time and speaks up when it does not hold.

Under `prefers-reduced-motion: reduce` nothing flies. The slide changes the way it would change without a morph. See the manual.

| Name | Vorgabe |
|---|---|
| `name` | – |

| Name | Vorgabe |
|---|---|
| `body` | – |
| `at` | `"1-"` |
| `duration` | `900` |
| `match` | `"auto"` |
| `inline` | `true` |

### 15.3.8 `pin`

```
pin(name, body)
```

A named piece inside a morph.

Shape matching pairs glyphs by their outline and, where that is not enough, by proximity. Most of the time that is right. Where it is not, because the 3 in `3x^4` is meant to become the 3 in `4 dot 3x^3` and another 3 sits in between, the piece gets a name, and the pairing follows that instead.

```
#morph(<term>)[$#pin(<faktor>)[3] x^#pin(<hoch>)[4]$]
… and on the next slide …
#morph(<term>)[$#pin(<hoch>)[4] dot #pin(<faktor>)[3] x^3$]
```

Matching names find each other before the shape is consulted; everything else works as before. A pin with no counterpart on the other slide falls back to shape matching without complaint.

The name is a string or a label.

### 15.3.9 `scene`

```
scene(
  ..parts,
  stops: (),
  tween: 8,
  start: auto,
  width: 100%,
  height: 190pt,
  duration: auto,
  enter: "fade",
  still: auto,
  steady: auto,
)
```

A drawing as a function of a value, with stops for the talk.

```
#scene(
  x => tangent-at(f, x),
  stops: (-3, 0, 1.5, 3),   // four stops, three steps
  tween: 8,                 // frames between two stops
)
```

This is manim's `ValueTracker` turned around. There a number changes while the film runs and the picture follows it; here Typst draws at compile time and a number can only change at a step. So the deck writes a function from a value to a picture and says at which values the talk stops. Typst renders every stop and the frames in between, and a keypress pulls the picture from one stop to the next.

`stops` are the values themselves, not 0.0 to 1.0 – that is the whole difference to `flipbook`. The scene takes `stops.len() - 1` steps: the first stop is there as soon as the scene appears, every further one costs a keypress.

A stop may be a tuple, and then the drawing function takes that many arguments: `(a, b) => …` with `stops: ((1, 1), (1, 3), (2, 3))`. What is lost against manim is that there several trackers may move independently; here everything travels from stop to stop together.

`tween` is the number of frames **between** two stops. With `tween: 0` the scene jumps from stop to stop and shows nothing in between.

`duration` is the time one pull from stop to stop takes, not the time of the entrance – the same separation `morph` draws, and for the same reason: one is a journey, the other a fade.

The scene stands in a box of a fixed size and every frame is clipped to it. The scene stands in a box of a fixed size and every frame is clipped to it. Unlike `build` the frames are **not** laid out on top of one another: they are drawings of different values and may legitimately come out different sizes, so one shared frame is the only arrangement in which the box itself does not jump.

The frames are measured all the same, and `steady` says what that measurement is for. A CeTZ canvas is as large as what it holds, so a frame wider than its neighbour puts the drawing somewhere else inside the box, and paging through it the whole picture travels while only one point should move. The package can see that and cannot correct it: `measure` answers with a size, never with where the ink lies inside it.

- `auto`, the default: the frames are measured and a finding is filed as a record. `presentation(drift: …)` decides what happens with the records – `"error"`, the default, stops at the end of the deck with all of them at once.
- `false`: the frames are meant to differ – a rectangle that grows, a number that counts up – and this scene is taken out of the check. It is not measured at all.
- `true`: this scene has to stand still, and it stops where it stands if it does not, whatever the deck says.

Measuring costs one more layout per frame, and a frame is a whole layout. Measured on a scene of 28 frames – four stops, eight frames between each pair, a CeTZ drawing of axes with ticks, a parabola of 61 points, a tangent, a dashed slope triangle and two labels: 434 ms without the measuring and 536 ms with it, so about 100 ms for the scene and 3.6 ms per frame. Only the browser branch pays it. On paper a scene is one still image, and a still image does not travel.

On paper the last stop is set, as with `alternatives`; `still` overrides that. The step cursor still runs there, so `info().step.total` names the same number in both outputs.

Under `prefers-reduced-motion: reduce` the frames in between fall away and the scene jumps from stop to stop. That is the package's rule everywhere else too: what stays is the destination, what goes is the travel.

| Name | Vorgabe |
|---|---|
| `..parts` | – |
| `stops` | `()` |
| `tween` | `8` |
| `start` | `auto` |
| `width` | `100%` |
| `height` | `190pt` |
| `duration` | `auto` |
| `enter` | `"fade"` |
| `still` | `auto` |
| `steady` | `auto` |

### 15.3.10 `scene-layer`

```
scene-layer(name, nr, body, enter: "fade")
```

Something that belongs to one particular stop of a scene.

A sentence beside it, a formula, a second drawing: it shares the step with its stop and therefore travels with it, without having to be linked.

```
#scene("derivative", x => tangent-at(f, x), stops: (-3, 0, 3))

#scene-layer("derivative", 2)[At the vertex the slope is zero.]
```

The scene has to stand **before** its layers in the source, because a layer looks up which step its stop was given. Standing after them, the package says so rather than quietly doing nothing.

A layer stays from its stop to the end of the slide, as `cue-layer` does: what was said at a stop goes on holding afterwards.

| Name | Vorgabe |
|---|---|
| `name` | – |
| `nr` | – |
| `body` | – |
| `enter` | `"fade"` |

### 15.3.11 `stagger`

```
stagger(
  ..items,
  start: auto,
  stride: 1,
  enter: "fade-up",
  duration: auto,
  easing: auto,
  stagger: 60,
  spacing: 0.65em,
  dim: false,
  morph: false,
  name: none,
)
```

Several things, one after another, one step apart.

```
#stagger[
  - The first point
  - The second
  - And the third
]
```

A bullet list is taken apart at its items; anything else is taken as it comes, one piece per argument. Where a list would be wrong – three cards side by side, say – hand the pieces over instead:

```
#stagger(card[One], card[Two], card[Three])
```

`start` is the step the first piece stands on, `auto` the next free one. `stride` is how many steps lie between two pieces: `2` leaves one out, and `0` puts them all on the same step, staggered only by `stagger`, which is the delay in milliseconds between one piece and the next. The two belong together – with `stride: 0` and `stagger: 60` a list arrives as a wave rather than as a sequence of keypresses.

`dim` leaves the pieces already shown standing, dimmed, instead of at full strength. `spacing` is the gap between them, `enter`, `duration` and `easing` are handed on to every piece unchanged.

`morph: true` lets each piece fly out of the one before it. Every piece stays where it is once it has arrived, so at a step change the piece set last is the source and the new one the target: the new line grows out of the line above while the line above stays put. That is a chain of transformations, line by line, on a single slide:

```
#stagger(morph: true, spacing: 14pt,
  $ x^2 + 6 x + 2 = 0 $,
  $ (x + 3)^2 - 7 = 0 $,
  $ x = -3 plus.minus sqrt(7) $,
)
```

A morph has no entrance, no easing curve and no dimmed rest, so `enter:`, `easing:` and `dim:` are refused rather than quietly dropped. `duration:` is read and is the time of the flight.

| Name | Vorgabe |
|---|---|
| `..items` | – |
| `start` | `auto` |
| `stride` | `1` |
| `enter` | `"fade-up"` |
| `duration` | `auto` |
| `easing` | `auto` |
| `stagger` | `60` |
| `spacing` | `0.65em` |
| `dim` | `false` |
| `morph` | `false` |
| `name` | `none` |

### 15.3.12 `stagger-layer`

```
stagger-layer(name, number, body, enter: "fade")
```

Something that belongs to one particular piece of a `stagger`.

A note in the margin, an annotation on a line of a calculation, a second drawing: it shares the step with its piece and therefore travels with it, without having to be counted.

```
#stagger(morph: "rewrite", ..lines)

#stagger-layer("rewrite", 2)[$| -2$]
```

The group needs a name for that – `name:` says it, and a `morph:` written as a name says it too, because then it is already there.

The stagger has to stand **before** its layers in the source, because a layer looks up which step its piece was given. Standing after them, the package says so rather than quietly doing nothing.

A layer stays from its piece to the end of the slide, as `cue-layer` and `scene-layer` do. And it stays out of a `morph: true` flight: the layer is its own element and carries no morph name, so what flies is the piece and the annotation merely appears beside it – which is what an annotation should do.

| Name | Vorgabe |
|---|---|
| `name` | – |
| `number` | – |

| Name | Vorgabe |
|---|---|
| `body` | – |
| `enter` | `"fade"` |

#### 15.3.13 `pause`

Everything after this appears a step later.

The short form for slides that simply unfold: no `anim` around anything, no step numbers.

```
== A slide
First this.
#pause
Then that.
```

It is read at the top level of the slide body, `#set` and `#show` rules included. Inside a grid cell, a table or a figure it is not seen: there the content is a field of an element, not part of the body. Reach for `anim` in those places instead.

## 15.4 Layouts

#### 15.4.1 `callout`

```
callout(
  body,
  title: auto,
  color: auto,
  radius: 7pt,
  inset: (x: 14pt, y: 11pt),
  width: 100%,
)
```

A highlighted key sentence: Beamer's `alertblock`.

The bar on the left marks it on every slide at a glance as „the thing to remember“, without it looking like a second box.

Labelled `<ts-callout>`, `<ts-callout-title>` and `<ts-callout-body>`. As in `card`, the surface goes through a `set` rule, so a label rule reaches `fill`, `stroke` and `radius`.

| Name | Vorgabe |
|---|---|
| `body` | – |
| `title` | `auto` |
| `color` | `auto` |
| `radius` | `7pt` |
| `inset` | `(x: 14pt, y: 11pt)` |
| `width` | `100%` |

### 15.4.2 `card`

```
card(
  body,
  title: none,
  number: none,
  color: auto,
  fill: auto,
  stroke: auto,
  radius: auto,
  inset: (x: 12pt, y: 10pt),
  width: 100%,
)
```

A named box: Beamer's `block`.

`title` sits in a colored bar above it, `number` additionally places a numbered disc in front. Without either, a plain box remains.

`color`, `fill` and `stroke` take the theme's values on `auto`: its primary color, its card background and its border.

No `clip: true`: Typst derives a clip path's identifier from the content, and the same box twice on a slide would produce the same identifier. The corners are therefore rounded by the bar itself.

Six labels for the six parts, so a deck can restyle them without touching the theme. `<ts-card>` is the box itself, and because its surface travels through a `set` rule rather than an argument, `set block(fill: ..)` on that label reaches it. `<ts-card-bar>` is the coloured tab, `<ts-card-title>` the caption, `<ts-card-disc>` the numbered disc, `<ts-card-number>` the numeral in it, `<ts-card-body>` the body.

| Name | Vorgabe |
|---|---|
| `body` | – |
| `title` | `none` |
| `number` | `none` |
| `color` | `auto` |
| `fill` | `auto` |
| `stroke` | `auto` |
| `radius` | `auto` |
| `inset` | `(x: 12pt, y: 10pt)` |
| `width` | `100%` |

### 15.4.3 `fit`

```
fit(
  body,
  width: auto,
  height: auto,
  wrap: true,
  grow: false,
  shrink: true,
)
```

Scale one block down to the room it has.

For the thing whose size the deck does not control: a wide table, a generated diagram, a list that came out of a data file. Without it such a block runs over the edge of the slide. In the PDF it is still to be seen standing there; in the browser the slide sits in a frame of fixed size and whatever reaches past it is cut away.

```
#slide[
  == Regression results
  #fit(wrap: false, my-table)
]
```

`wrap: false` because the block is a table. Everything that lays itself out in columns has to be measured as it stands; see below.

The block is measured against the place it stands in and scaled geometrically, so it keeps its proportions and what stands around it counts with its new size. No factor is given by hand.

**Width first, then smaller.** The block is offered the full width before it is measured, so a paragraph or a list breaks into the space instead of shrinking, and only what is still too tall afterwards is scaled. A table, a chart or a drawing would rearrange its own columns instead, which changes the picture rather than its size; `wrap: false` measures such a block exactly as it stands.

**It only shrinks.** `grow: true` also blows a block up that is smaller than its place, for the one large number that is meant to fill the slide. `shrink: false` takes the shrinking away and leaves only the growing.

`width` and `height` take `auto`, a length or a ratio. `auto` is the whole place. On `height: auto` the block takes what is left over below whatever stands above it on the slide, so a fit under two bullet points reckons with the bullet points. That has a flip side wherever something encloses the fit: inside a `card` the box becomes slide-tall, is cut off at the bottom, and **whatever follows the card falls off the slide** – measured in both outputs. It is the `1fr` doing that, not the scaling: a `card` around a bare `block(height: 1fr)` behaves the same. Give `height:` explicitly inside a card, and the fit reckons with that instead.

**No reveal inside.** Two things do not survive being measured. A `pause` is found by walking the slide body, and a fitted block is a closure that walk cannot enter: measured on a slide carrying two pauses, the step count fell from three to one and nothing said so. And a measured block has no height to reckon against, which is the axis on which a tracked element resolves its size and reserves the room for its marker: measured, an `anim` inside a fit was not scaled at all and ran off the bottom of the slide. `fit` stops with a message instead, for `pause`, `anim`, `stagger`, `alternatives`, `morph`, `tiles`, `video`, `embed`, `flipbook`, `build`, `scene`, `camera` and `cue`, in both outputs. Fit what stands inside the reveal rather than fitting around it: `anim(fit(my-table))` works, `fit(anim(my-table))` is the error.

`speaker-note` and `bridge-job` are allowed inside a fit. They settle no geometry, and a `measure` commits no state, so both were measured to arrive exactly once. The other direction is the one that does not work: a note made only of a `fit` carries no text, and `speaker-note` refuses it.

The geometry is adapted from mosaic, which adapted Touying 0.7.4, which credits it to Andreas Kröpelin (Polylux PR 91) and to ntjess.

| Name | Vorgabe |
|---|---|
| `body` | – |
| `width` | `auto` |
| `height` | `auto` |
| `wrap` | `true` |
| `grow` | `false` |
| `shrink` | `true` |

### 15.4.4 `side-by-side`

```
side-by-side(
  ..parts,
  split: (1.25fr, 1fr),
  gutter: 18pt,
  align: horizon,
  equal: false,
)
```

Two or more columns side by side.

The name comes from Touying and Polylux, which chose it independently of each other.

`split` takes the column widths; the default gives the first column a bit more, because that is usually where the illustration sits and the text on the right.

| Name | Vorgabe |
|---|---|
| `..parts` | – |
| `split` | `(1.25fr, 1fr)` |
| `gutter` | `18pt` |
| `align` | `horizon` |
| `equal` | `false` |

### 15.4.5 `statement`

```
statement(
  body,
  size: 1.6em,
  color: none,
  above: 0.6em,
  below: 0.6em,
)
```

A large statement in the middle: the formula that matters.

Explicitly demands the full width: a tracked element becomes as wide as its content, and a bare `align(center, …)` inside it would have no room to center in and would sit unchanged on the left.

Labelled `<ts-statement>`. `size` is measured in `em`, so a `set text(size: …)` from a label rule multiplies rather than replaces it.

| Name | Vorgabe |
|---|---|
| `body` | – |
| `size` | `1.6em` |
| `color` | `none` |
| `above` | `0.6em` |
| `below` | `0.6em` |

### 15.4.6 `tiles`

```
tiles(
  ..items,
  columns: auto,
  gutter: 14pt,
  row-gutter: auto,
  at: auto,
  stride: 1,
  stagger: 0,
  enter: "fade-up",
  duration: auto,
  easing: auto,
  align: top + left,
)
```

A tile grid that staggers itself.

Each tile appears one step after the previous one: that is the reason this function exists. By hand that means an `anim` per tile with an incremented number or delay.

`at` behaves as in `anim`: `auto` takes the next free step. `stride: 0` makes all tiles appear on the same step and staggers only through `stagger`, in milliseconds; a wave then runs through the grid.

`duration` and `easing` are those of `anim` and apply to every tile alike: one grid moves as one thing. `auto` is the presentation's duration and the package's own curve. Without them a deck that wanted a slower tile or a curve with a swing had to leave the grid and write the `anim`s out by hand, which is the very work `tiles` exists to save.

| Name | Vorgabe |
|---|---|
| `..items` | – |
| `columns` | `auto` |
| `gutter` | `14pt` |
| `row-gutter` | `auto` |
| `at` | `auto` |
| `stride` | `1` |
| `stagger` | `0` |
| `enter` | `"fade-up"` |
| `duration` | `auto` |
| `easing` | `auto` |
| `align` | `top + left` |

## 15.5 Themes

### 15.5.1 `theme`

```
theme(
  paper: rgb("#fafafa"),
  ink: black,
  strong: rgb("#23303f"),
  accent: rgb("#eb5e28"),
  muted: luma(45%),
  surface: white,
  border: luma(84%),
  inverted: false,
  font: none,
  title-font: none,
  size: 24pt,
  title-size: 31pt,
  weight: "bold",
  tracking: 0pt,
  header: "band",
  title-fill: p => lesbar(p.strong, white, p.paper, p.ink),
  rule-size: 0pt,
  rule-fill: none,
  head-gap: 20pt,
  foot-gap: 24pt,
  band-height: 66pt,
  footer: "fraction",
  footer-rule: 0pt,
  progress: "bar",
  box: "bar",
  title-slide: band-title-slide,
  section: band-section,
)
```

Builds a theme. Without an argument it produces the default appearance: `themes.default` is exactly that.

The entries in groups: colors (`paper ink strong accent muted surface border inverted`), typography (`font title-font size title-size weight tracking`), the ordinary slide (`header title-fill rule-size rule-fill head-gap foot-gap band-height box footer footer-rule progress`) and the two whole pictures (`title-slide`, `section`).

The eight colors are also the entries a **palette** may carry, and that is how a theme is recolored without touching the rest of it. Two of the entries above are colors that are not palette entries, `title-fill` and `rule-fill`, and each may be written either as a color or as a function of the palette, `p => p.strong`. A color stays what it is under every palette; a function is asked again whenever one is applied, and that is what lets the title of a light theme follow into the dark. `rule-fill: none` means the accent and follows it.

Font sizes: measured against Beamer and Metropolis, where the body text takes up around 3.0% of the slide width and the title 3.9%. The earlier 19pt/23pt were at 2.3% and 2.7%: noticeably smaller than what you would want to read from the back row.

What the theme draws also carries labels, and those are the second way to reach it. Names follow one scheme, place first and part second. On an ordinary slide `ts-slide-ground`, `ts-slide-header-band`, `ts-slide-header-text`, `ts-slide-header-rule`, `ts-slide-title`, `ts-slide-title-rule`, `ts-slide-footer`,

`ts-slide-number`, `ts-slide-footer-rule`, `ts-slide-progress` and `ts-slide-progress-track`; on the title slide `ts-title-slide-ground`, `ts-title-slide-band`, `ts-title-slide-title`, `ts-title-slide-subtitle`, `ts-title-slide-rule` and `ts-title-slide-byline`; on the section slide `ts-section-slide-ground`, `ts-section-slide-bar`, `ts-section-slide-title`, `ts-section-slide-rule` and `ts-section-slide-parent`. A `show` rule on one of them changes type or fill without a key having to exist for it; the keys below stay what they are and keep the arrangement.

The last of those is the line naming the sections a deeper section hangs under. It exists only from the second structure level on, so a deck at the default `slide-level: 2` never draws it.

A theme that brings its own `title-slide` or `section` function draws none of those labels, and nothing warns about it. Such a `section` function receives the section record, and there `s.depth` is the heading level and `s.parents` are the titles above it, outermost first. A function that reads neither draws every level alike; nothing breaks, the hierarchy is simply not shown.

Comments must NOT go into the parameter list: tidy splits it at the commas and expects a colon in every piece; the API reference breaks on that.

| Name | Vorgabe |
|---|---|
| `paper` | `rgb("#fafafa")` |
| `ink` | `black` |
| `strong` | `rgb("#23303f")` |
| `accent` | `rgb("#eb5e28")` |
| `muted` | `luma(45%)` |
| `surface` | `white` |
| `border` | `luma(84%)` |
| `inverted` | `false` |
| `font` | `none` |
| `title-font` | `none` |
| `size` | `24pt` |
| `title-size` | `31pt` |
| `weight` | `"bold"` |
| `tracking` | `0pt` |
| `header` | `"band"` |
| `title-fill` | `p => lesbar(p.strong, white, p.paper, p.ink)` |
| `rule-size` | `0pt` |
| `rule-fill` | `none` |
| `head-gap` | `20pt` |
| `foot-gap` | `24pt` |
| `band-height` | `66pt` |
| `footer` | `"fraction"` |
| `footer-rule` | `0pt` |
| `progress` | `"bar"` |
| `box` | `"bar"` |
| `title-slide` | `band-title-slide` |
| `section` | `band-section` |

### 15.5.2 `themes`

The bundled themes: `themes.default`, `themes.lesson`, `themes.night`, `themes.plain`, `themes.editorial`.

They are made for different occasions, not the same slide in five colors: the title sits sometimes in a bar, sometimes free, sometimes under a line; the progress indicator grows, or is missing entirely.

## 15.6 Palettes

### 15.6.1 `contrast`

```
contrast(a, b)
```

The WCAG contrast ratio of two colours, a number from 1 to 21.

```
#contrast(black, white)                // 21
#contrast(rgb("#767b84"), white)       // 4.2539
```

The reference numbers WCAG 2 names: 4.5 for body text, 3.0 for large text and for lines, bars and other shapes that are not text. The package uses them in that sense in `palette-report`.

Alpha is ignored. A translucent colour is measured as if it were opaque, which is not what it looks like on the slide.

### 15.6.2 `palette-report`

```
palette-report(p)
```

Measure a palette against the contrast contract and report every pair.

```
#for f in palette-report(palettes.dark) [
  #f.pair: #calc.round(f.ratio, digits: 2) (wants #f.min) #f.ok \
]
```

One dictionary per pair, with `pair`, `ratio`, `min`, `ok` and `role`. It only measures and never changes a colour.

This is a report, not a gate. Only the five bundled palettes are actually held to the contract, by an assertion in this file; a palette written in a deck faces no such check, and none of the five bundled **themes** does either. Four of the five do not pass it, and that is deliberate rather than overlooked. See the manual.

### 15.6.3 `palettes`

The bundled palettes.

Five, one per colour world the package already had, and each of them composes with every theme: `themes.lesson` in `palettes.dark` is still the lesson design, only dark.

- `light` is exactly the default theme's colours, so `themes.default` with it is a no-op.
- `mono` is `themes.plain`'s greyscale, with two greys moved so it passes the contract.
- `textbook` is `themes.lesson`'s measured textbook colours, with `muted` moved for the same reason.
- `parchment` is `themes.editorial`'s laid paper, with `accent` and `muted` moved for the same reason.
- `dark` is `themes.night`'s dark ground with a deeper accent, because night's own cyan does not survive an inverted slide.

The six numbers that were moved, and why, are in the comments below.

## 15.7 Media and embeds

### 15.7.1 `embed`

```
embed(
  url: none,
  html: none,
  width: 100%,
  height: 200pt,
  at: "1-",
  enter: "fade",
  bridge: none,
  zoom: true,
  style: true,
  fallback: none,
  link: none,
  label: auto,
)
```

Arbitrary web content in a sandboxed frame.

`bridge` names the element so step jobs can be sent to it: that is how `geogebra` drives its applet, and how a companion package of your own would drive anything else, without the core knowing what is inside.

`fallback` and `link` only take effect in paged output; in the browser the embedded document itself stands there.

`style` gives a document passed as `html` the deck's basic style: it fills the frame, is transparent, and carries the running text size. Switched off, the frame is a blank browser page again.

| Name | Vorgabe |
|---|---|
| `url` | `none` |
| `html` | `none` |
| `width` | `100%` |
| `height` | `200pt` |
| `at` | `"1-"` |
| `enter` | `"fade"` |
| `bridge` | `none` |
| `zoom` | `true` |
| `style` | `true` |
| `fallback` | `none` |
| `link` | `none` |
| `label` | `auto` |

### 15.7.2 `flipbook`

```
flipbook(
  render,
  frames: 24,
  fps: 30,
  width: 200pt,
  height: 150pt,
  loop: true,
  pingpong: false,
  at: "1-",
  enter: "fade",
  still: auto,
)
```

Animation drawn by Typst, frame by frame.

`render` receives `t` running from 0.0 to 1.0. Every frame is rendered by Typst: CeTZ, Fletcher, equations, anything Typst can do. The frames sit in the file as SVG and stay sharp at any size.

Under `prefers-reduced-motion: reduce` it does not play. It stands on its last frame without `loop` and without `pingpong`, and on frame zero otherwise. See the manual.

| Name | Vorgabe |
|---|---|
| `render` | – |
| `frames` | `24` |
| `fps` | `30` |
| `width` | `200pt` |
| `height` | `150pt` |
| `loop` | `true` |
| `pingpong` | `false` |
| `at` | `"1-"` |
| `enter` | `"fade"` |
| `still` | `auto` |

### 15.7.3 `video`

```
video(
  src,
  width: 100%,
  height: 200pt,
  poster: none,
  autoplay: true,
  loop: false,
  muted: true,
  controls: false,
  radius: 0pt,
  at: "1-",
  enter: "fade",
)
```

A real HTML5 video over the slide.

Without a `poster:` the placeholder on paper is labelled `<ts-media-poster>`.

| Name | Vorgabe |
|---|---|
| `src` | – |

| Name | Vorgabe |
|---|---|
| `width` | `100%` |
| `height` | `200pt` |
| `poster` | `none` |
| `autoplay` | `true` |
| `loop` | `false` |
| `muted` | `true` |
| `controls` | `false` |
| `radius` | `0pt` |
| `at` | `"1-"` |
| `enter` | `"fade"` |

## 15.8 The bridge

### 15.8.1 `bridge-job`

```
bridge-job(target, payload, at: "1-")
```

Send a job to a bridged element.

- `target` is the `bridge` name given to `embed`.
- `at` is a step selector, resolved exactly as for `anim`: `auto` takes the next free step, an integer counts as „from this step on“. The default is `"1-"`, because most jobs set the document up on slide entry.

  A job does not move the step cursor: an applet's tween and the bullet explaining it usually belong on the **same** step, not one after the other.
- `payload` is a dictionary. Its meaning is entirely up to the document on the other side. The core passes it through unread.

When paging backwards or entering a slide the runtime replays the whole run from its start with a `reset` flag, so jobs should be repeatable.

**The document has to announce itself.** Nothing is sent to a frame that has not said hello. The runtime marks a frame live only after receiving

```
parent.postMessage({ typstage: 1, ready: 1 }, "*");
```

from it. Both fields are needed: the runtime drops every message without `typstage: 1` before it ever looks at `ready`. Miss this and the jobs simply never arrive. There is no error, the applet just sits there.

| Name | Vorgabe |
|---|---|
| `target` | – |
| `payload` | – |
| `at` | `"1-"` |

### 15.8.2 `bridge-targets`

```
bridge-targets()
```

The bridge names on the current slide, in the order they appear.

This is how a companion package can leave the applet unnamed: with exactly one bridged element on the slide there is nothing to choose between.

Must be called in a context. Duplicates are dropped, because a tracked element is laid out twice, once in the background and once as its sprite, so it announces itself twice.

## 15.9 GeoGebra

### 15.9.1 `geogebra`

```
geogebra(
  ..name,
  id: "ggb",
  material: none,
  app: "classic",
  perspective: "G",
  language: none,
  grid: auto,
  axes: auto,
  seamless: true,
  background: auto,
  animation-button: false,
  ,
  pan: false,
  font-size: 17,
  codebase: "https://www.geogebra.org/apps/",
  width: 100%,
  height: 330pt,
  at: "1-",
  fallback: none,
  link: none,
)
```

A GeoGebra applet in the slide.

- `seamless` takes the frame off the applet and puts its drawing area in the slide's colour, so it no longer looks like a window of its own.
- `fallback` and `link` only take effect in the PDF: there is no applet on paper, so what stands there is either a drawing of your own or at least the way to the live one.

The `id` is only needed when a slide holds more than one applet — the commands then say which one they mean. A single applet needs no name.

A `.ggb` file cannot be embedded: Typst has no base64 encoding, and without it the file content never reaches the HTML. Build the construction with `ggb-run` or load it through `material`.

| Name | Vorgabe |
|---|---|
| `..name` | – |
| `id` | `"ggb"` |
| `material` | `none` |
| `app` | `"classic"` |
| `perspective` | `"G"` |
| `language` | `none` |
| `grid` | `auto` |
| `axes` | `auto` |
| `seamless` | `true` |
| `background` | `auto` |
| `animation-button` | `false` |
| | – |

| Name | Vorgabe |
|---|---|
| `pan` | `false` |
| `font-size` | `17` |
| `codebase` | `"https://www.geogebra.org/apps/"` |
| `width` | `100%` |
| `height` | `330pt` |
| `at` | `"1-"` |
| `fallback` | `none` |
| `link` | `none` |

### 15.9.2 `ggb-animate`

```
ggb-animate(
  ..objects,
  target: auto,
  at: "1-",
  speed: none,
  playing: true,
  trace: (),
)
```

GeoGebra's own animation — it runs back and forth without end.

| Name | Vorgabe |
|---|---|
| `..objects` | – |
| `target` | `auto` |
| `at` | `"1-"` |
| `speed` | `none` |
| `playing` | `true` |
| `trace` | `()` |

### 15.9.3 `ggb-hide`

```
ggb-hide(..objects, target: auto, at: "1-")
```

Hide objects — the counterpart to `ggb-show`.

| Name | Vorgabe |
|---|---|
| `..objects` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### 15.9.4 `ggb-run`

```
ggb-run(..commands, target: auto, at: "1-")
```

Run GeoGebra commands.

GeoGebra's scripting commands — `SetColor`, `SetValue` and relatives — are **not** accepted by `evalCommand` and would come to nothing here. That is what `ggb-style`, `ggb-set`, `ggb-show` and `ggb-hide` are for: they reach for the JavaScript interface, which can do it.

| Name | Vorgabe |
|---|---|
| `..commands` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### 15.9.5 `ggb-set`

```
ggb-set(values, target: auto, at: "1-")
```

Set values in the applet.

| Name | Vorgabe |
|---|---|
| `values` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### 15.9.6 `ggb-show`

```
ggb-show(..objects, target: auto, at: "1-")
```

Reveal objects that were hidden before.

| Name | Vorgabe |
|---|---|
| `..objects` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### 15.9.7 `ggb-style`

```
ggb-style(
  ..objects,
  target: auto,
  at: "1-",
  color: none,
  thickness: none,
  line-style: none,
  filling: none,
  point-size: none,
  trace: none,
  label: none,
  label-mode: none,
  fixed: none,
  caption: none,
  layer: none,
  position: none,
)
```

Appearance — `color` takes a Typst colour, so the construction carries the colours of your slides instead of GeoGebra's palette.

| Name | Vorgabe |
|---|---|
| `..objects` | – |
| `target` | `auto` |
| `at` | `"1-"` |

| Name | Vorgabe |
|---|---|
| `color` | `none` |
| `thickness` | `none` |
| `line-style` | `none` |
| `filling` | `none` |
| `point-size` | `none` |
| `trace` | `none` |
| `label` | `none` |
| `label-mode` | `none` |
| `fixed` | `none` |
| `caption` | `none` |
| `layer` | `none` |
| `position` | `none` |

### 15.9.8 `ggb-tween`

```
ggb-tween(
  ..name,
  target: auto,
  to: 1.0,
  from: none,
  at: 1,
  duration: 650,
  easing: "ease-in-out",
)
```

Once from A to B — the construction draws itself.

GeoGebra's own animation runs back and forth for ever. Drawing needs the opposite — once from 0 to 1 and then stop — so here the browser counts the value up frame by frame. Build an object that depends on it and it grows along: a segment whose endpoint travels, an arc whose angle follows.

`at` points at **one** step — that is where it is drawn. On the steps after it the value simply sits at its target, so jumping back shows the finished drawing instead of the movement a second time.

| Name | Vorgabe |
|---|---|
| `..name` | – |
| `target` | `auto` |
| `to` | `1.0` |
| `from` | `none` |
| `at` | `1` |
| `duration` | `650` |
| `easing` | `"ease-in-out"` |

### 15.9.9 `ggb-view`

```
ggb-view(
  target: auto,
  at: "1-",
  x: none,
  y: none,
  grid: none,
  axes: none,
)
```

Viewport, grid, axes.

| Name | Vorgabe |
|---|---|
| `target` | `auto` |
| `at` | `"1-"` |
| `x` | `none` |
| `y` | `none` |
| `grid` | `none` |
| `axes` | `none` |

**`desmos`**

```
desmos(
  ..name,
  id: "dsm",
  api-key: none,
  version: "1.11",
  expressions: (:),
  bounds: none,
  ,
  expression-list: false,
  settings-menu: false,
  zoom-buttons: false,
  keypad: false,
  pan: false,
  grid: auto,
  axes: auto,
  axis-numbers: auto,
  seamless: true,
  background: auto,
  width: 100%,
  height: 330pt,
  at: "1-",
  fallback: none,
  link: none,
)
```

Ein Desmos-Graph auf der Folie.

- `api-key` ist Pflicht. Desmos gibt sein Skript ohne Schlüssel nicht heraus – gemessen antwortet der Server dann mit 403. Zum Ausprobieren gibt es `demo-key`; wer damit vorträgt, trägt Desmos' Konsolenwarnung mit sich herum und arbeitet außerhalb dessen, wofür der Schlüssel gedacht ist.
- `expressions` ist das Startbild als Wörterbuch von `id` nach `LaTeX`. Ein Ausdruck mit `=` ist ein Regler, einer ohne eine Kurve; das entscheidet Desmos.
- `bounds` ist der Ausschnitt als `(links, rechts, unten, oben)`.

| Name | Vorgabe |
|---|---|
| `..name` | – |
| `id` | `"dsm"` |
| `api-key` | `none` |
| `version` | `"1.11"` |
| `expressions` | `(:)` |
| `bounds` | `none` |
| | – |
| `expression-list` | `false` |
| `settings-menu` | `false` |
| `zoom-buttons` | `false` |
| `keypad` | `false` |
| `pan` | `false` |
| `grid` | `auto` |
| `axes` | `auto` |
| `axis-numbers` | `auto` |
| `seamless` | `true` |
| `background` | `auto` |
| `width` | `100%` |
| `height` | `330pt` |
| `at` | `"1-"` |
| `fallback` | `none` |
| `link` | `none` |

### `dsm-animate`

```
dsm-animate(
  ..ids,
  target: auto,
  at: "1-",
  playing: true,
  min: none,
  max: none,
  step: none,
)
```

Desmos' eigene Regleranimation an- oder abschalten.

Sie läuft mit Desmos' Geschwindigkeit und ohne Ziel. Wer von einer Zahl zu einer anderen will und dann stehenbleiben, nimmt `dsm-tween`.

| Name | Vorgabe |
|---|---|
| `..ids` | – |
| `target` | `auto` |
| `at` | `"1-"` |
| `playing` | `true` |
| `min` | `none` |
| `max` | `none` |
| `step` | `none` |

### `dsm-expr`

```
dsm-expr(..objects, target: auto, at: "1-")
```

Ganze Ausdrucksobjekte durchreichen, für alles, was `dsm-set` nicht kann – Farbe, Reglergrenzen und Stil in einem Zug.

| Name | Vorgabe |
|---|---|
| `..objects` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### `dsm-hide`

```
dsm-hide(..ids, target: auto, at: "1-")
```

Ausdrücke verbergen. Sie bleiben im Graphen und rechnen weiter mit.

| Name | Vorgabe |
|---|---|
| `..ids` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### `dsm-remove`

```
dsm-remove(..ids, target: auto, at: "1-")
```

Ausdrücke entfernen.

| Name | Vorgabe |
|---|---|
| `..ids` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### `dsm-set`

```
dsm-set(values, target: auto, at: "1-")
```

Ausdrücke setzen oder ändern – ein Wörterbuch von `id` nach `LaTeX`.

Dieselbe `id` noch einmal ersetzt den Ausdruck, statt einen zweiten anzulegen. So bewegt sich eine Kurve über die Schritte.

| Name | Vorgabe |
|---|---|
| `values` | – |
| `target` | `auto` |
| `at` | `"1-"` |

### `dsm-show`

```
dsm-show(..ids, target: auto, at: "1-")
```

Ausdrücke zeigen.

| Name | Vorgabe |
|---|---|
| `..ids` | – |
| `target` | `auto` |
| `at` | `"1-"` |

#### `dsm-style`

```
dsm-style(
  ..ids,
  target: auto,
  at: "1-",
  color: none,
  line-style: none,
  line-width: none,
  line-opacity: none,
  point-style: none,
  point-size: none,
  fill: none,
  fill-opacity: none,
  label: none,
  show-label: none,
  drag-mode: none,
)
```

Aussehen eines Ausdrucks. Die Schlüssel sind Desmos' eigene.

| Name | Vorgabe |
|---|---|
| `..ids` | – |
| `target` | `auto` |
| `at` | `"1-"` |
| `color` | `none` |
| `line-style` | `none` |
| `line-width` | `none` |
| `line-opacity` | `none` |
| `point-style` | `none` |
| `point-size` | `none` |
| `fill` | `none` |
| `fill-opacity` | `none` |
| `label` | `none` |
| `show-label` | `none` |
| `drag-mode` | `none` |

#### `dsm-tween`

```
dsm-tween(
  name,
  target: auto,
  to: 1.0,
  from: none,
  at: 1,
  duration: 600,
  easing: "ease",
)
```

Einen Regler von einer Zahl zur anderen ziehen, in einer gegebenen Zeit.

Zwei Aufträge, wie bei `ggb-tween` nebenan, und der zweite ist der wichtigere: Die Bewegung liegt auf **genau** diesem Schritt, und ab dem nächsten steht der Endwert einfach da. Ohne das liefe sie auf jedem weiteren Schritt der Folie noch einmal an – gemessen sprang der Regler beim Weiterblättern von 3 zurück auf 0,75 und wuchs erneut, weil ein ganzzahliges `at` zu „ab diesem Schritt“ wird und nicht zu „auf diesem“.

Auf Papier und beim Zurückblättern steht das Ergebnis sofort da: eine Bewegung, die niemand sieht, ist keine.

| Name | Vorgabe |
|---|---|
| `name` | – |
| `target` | `auto` |
| `to` | `1.0` |
| `from` | `none` |
| `at` | `1` |
| `duration` | `600` |
| `easing` | `"ease"` |

#### `dsm-view`

```
dsm-view(
  target: auto,
  at: "1-",
  bounds: none,
  grid: none,
  axes: none,
  axis-numbers: none,
  degrees: none,
  polar: none,
)
```

Der Ausschnitt, als `(links, rechts, unten, oben)`, und die Grapheinstellungen.

| Name | Vorgabe |
|---|---|
| `target` | `auto` |
| `at` | `"1-"` |
| `bounds` | `none` |
| `grid` | `none` |
| `axes` | `none` |
| `axis-numbers` | `none` |
| `degrees` | `none` |
| `polar` | `none` |

#### `demo-key`

Der Demo-Schlüssel aus Desmos’ eigener Dokumentation.

Er ist ausdrücklich zum Ausprobieren gedacht, nicht für den Betrieb. Das geladene Skript sagt es selbst in der Konsole: „This page is using the Desmos API with a trial key suitable for prototyping, not for commercial use.“ Einen eigenen gibt es über https://www.desmos.com/my-api.

## 15.10 Measurements, colours, runtime files

### 15.10.1 `dark`

The default palette. Override it by wrapping the presentation in your own document template. See `style` on `presentation`.

### 15.10.2 `runtime-files`

The runtime files, ready to be written next to the HTML.

Typst cannot create files. Whoever uses `assets: "split"` or a CDN writes them out once. The content comes from here so the copies cannot drift.

### 15.10.3 `runtime-version`

Version of the runtime. It goes into the asset file names so a CDN can hold several releases side by side and no browser serves a stale one from cache.

### 15.10.4 `slide-width`

Default slide geometry. 16:9 on an A4-width canvas, so a slide and a handout page carry text at the same physical size. `presentation` takes `width`, `height` and `margin` to override them.