

CeTZ

ein Typst
Zeichenpaket

Johannes Wolf
fenjalien

Version 0.2.2

1 Introduction	3	3.7 Projection	32
2 Usage	3	3.7.1 ortho	32
2.1 CeTZ Unique Argument Types	3	3.7.2 on-xy	33
2.2 Anchors	3	3.7.3 on-xz	34
2.2.1 Named	4	3.7.4 on-yz	34
2.2.2 Border	4	3.8 Bodies	35
2.2.3 Path	4	3.8.1 prism	35
3 Draw Function Reference	5	4 Coordinate Systems	36
3.1 Canvas	5	4.1 XYZ	36
3.1.1 canvas	5	4.2 Previous	37
3.2 Styling	5	4.3 Relative	37
3.2.1 Marks	6	4.4 Polar	37
3.3 Shapes	10	4.5 Barycentric	38
3.3.1 circle	10	4.6 Anchor	38
3.3.2 circle-through	10	4.7 Tangent	39
3.3.3 arc	11	4.8 Perpendicular	39
3.3.4 arc-through	12	4.9 Interpolation	40
3.3.5 mark	13	4.10 Function	41
3.3.6 line	14	5 Libraries	43
3.3.7 grid	14	5.1 Tree	43
3.3.8 content	15	5.1.1 tree	43
3.3.9 rect	16	5.2 Palette	44
3.3.10 bezier	17	5.2.1 new	45
3.3.11 bezier-through	18	5.2.2 List of predefined palettes	45
3.3.12 catmull	19	5.3 Angle	46
3.3.13 hobby	20	5.3.1 angle	46
3.3.14 merge-path	21	5.3.2 right-angle	47
3.4 Grouping	22	5.4 Decorations	48
3.4.1 hide	22	5.4.1 Braces	48
3.4.2 floating	22	5.4.2 brace	48
3.4.3 intersections	23	5.4.3 flat-brace	49
3.4.4 group	23	5.4.4 Path Decorations	50
3.4.5 scope	24	5.4.5 zigzag	51
3.4.6 anchor	25	5.4.6 coil	52
3.4.7 copy-anchors	25	5.4.7 wave	52
3.4.8 set-ctx	26	6 Advanced Functions	53
3.4.9 get-ctx	26	6.1 Coordinate	54
3.4.10 for-each-anchor	27	6.1.1 resolve	54
3.4.11 on-layer	27	6.2 Styles	54
3.5 Utility	29	6.2.1 resolve	54
3.5.1 assert-version	29	6.2.2 Default Style	56
3.6 Transformations	30	7 Creating Custom Elements	57
3.6.1 set-transform	30	8 Internals	57
3.6.2 rotate	30	8.1 Context	57
3.6.3 translate	30	8.2 Elements	57
3.6.4 scale	31		
3.6.5 set-origin	31		
3.6.6 move-to	32		
3.6.7 set-viewport	32		

1 Introduction

This package provides a way to draw onto a canvas using a similar API to [Processing](#) but with relative coordinates and anchors from [TikZ](#). You also won't have to worry about accidentally drawing over other content as the canvas will automatically resize. And remember: up is positive!

The name CeTZ is a recursive acronym for “CeTZ, ein Typst Zeichenpaket” (german for “CeTZ, a Typst drawing package”).

2 Usage

This is the minimal starting point:

```
#import "@preview/cetz:0.2.2"
#cetx.canvas({
  import cetx.draw: *
  ...
})
```

Note that draw functions are imported inside the scope of the canvas block. This is recommended as some draw functions override Typst's functions such as `line`.

2.1 CeTZ Unique Argument Types

Many CeTZ functions expect data in certain formats which we will call types. Note that these are actually made up of Typst primitives.

coordinate Any coordinate system. See [coordinate-systems](#).

number Any of `float`, `integer` or `length`.

style Named arguments (or a dictionary if used for a single argument) of style key-values.

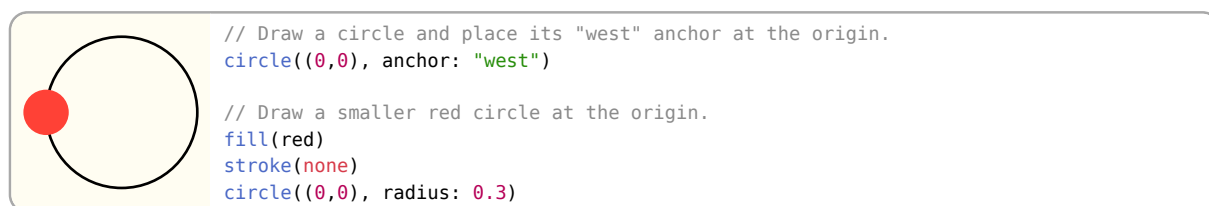
context A CeTZ context object that holds internal state.

vector A three element array of `float`s

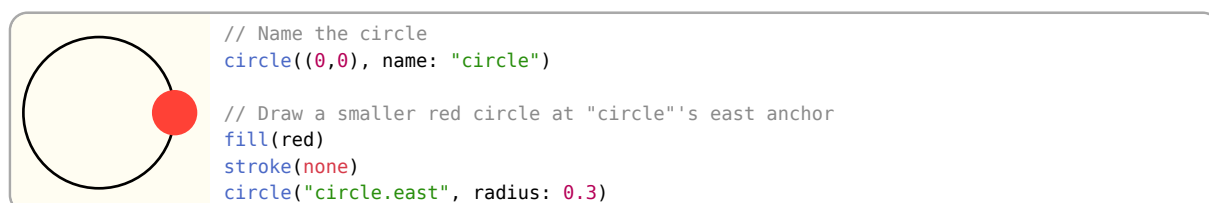
2.2 Anchors

You can refer to a position relative to an element by using its anchors. Anchors come in several different variations but can all be used in two different ways.

The first is by using the anchor argument on an element. When given, the element will be translated such that the given anchor will be where the given position is. This is supported by all elements that have the anchor argument.



The second is by using anchor coordinates. You must first give the element a name by passing a string to its name argument, you can then use its anchors to place other elements, see [Section 4.6](#) for more usage. Note this is only available for elements that have a name argument.



Note that all anchors are transformed along with the element.

2.2.1 Named

Named anchors are normally unique to the type of element, such as a bezier curve's control points. Other anchor variants specify their own named anchors that are available to all elements that support the anchor variant.

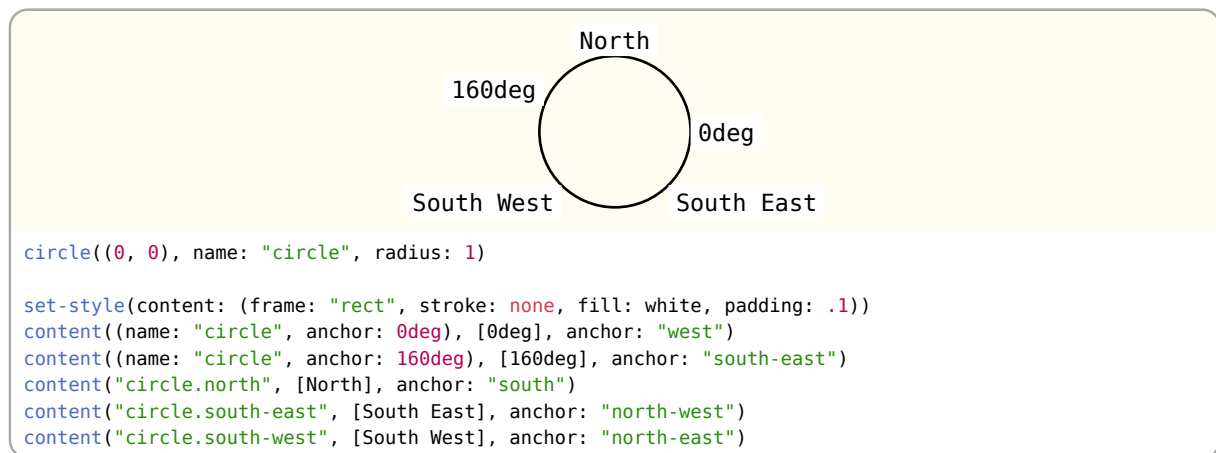
All elements also have a “default” named anchor, it always refers to another anchor on the element.

2.2.2 Border

A border anchor refers to a point on the element's border where a ray is cast from the element's center at a given angle and hits the border.

They are given as angles where 0deg is towards the right and 90deg is up.

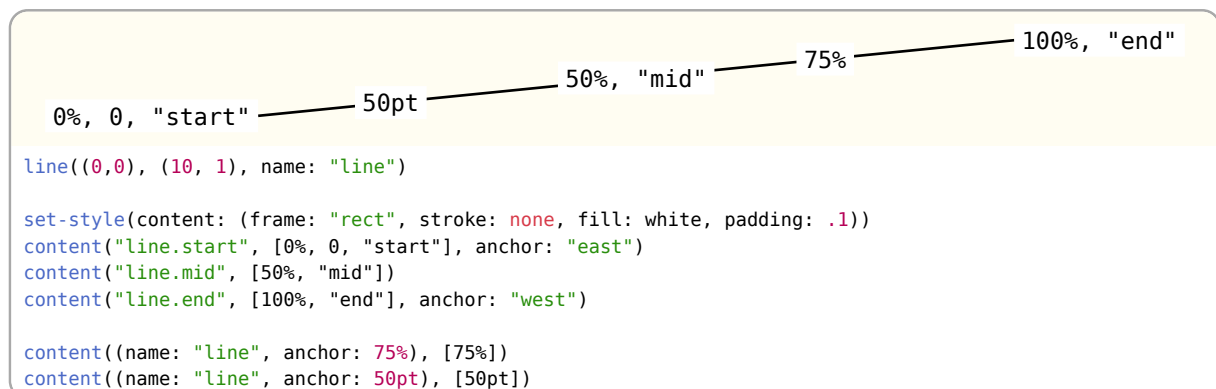
Border anchors also specify named compass directions such as “north,” “north-east,” etc. Border anchors also specify a “center” named anchor which is where the ray cast originates from.



2.2.3 Path

A path anchor refers to a point along the path of an element. They can be given as either a **number** for an absolute distance along the path, or a **ratio** for a relative distance along the path.

Path anchors also specify three anchors “start,” “mid,” and “end”.



3 Draw Function Reference

3.1 Canvas

3.1.1 canvas

Sets up a canvas for drawing on.

The default transformation matrix of the canvas is set to: $\begin{pmatrix} 1 & 0 & -0.5 & 0 \\ 0 & -1 & 0.5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Parameters

```
canvas(
  length: length ratio,
  debug: bool,
  background: none color,
  body: none array element
) -> content
```

length: length or ratio

Default: 1cm

Used to specify what 1 coordinate unit is. If given a ratio, that ratio is relative to the containing elements width!

debug: bool

Default: false

Shows the bounding boxes of each element when true.

background: none or color

Default: none

A color to be used for the background of the canvas.

body: none or array or element

A code block in which functions from draw.typ have been called.

3.2 Styling

You can style draw elements by passing the relevant named arguments to their draw functions. All elements that draw something have stroke and fill styling unless said otherwise.

fill: color or none

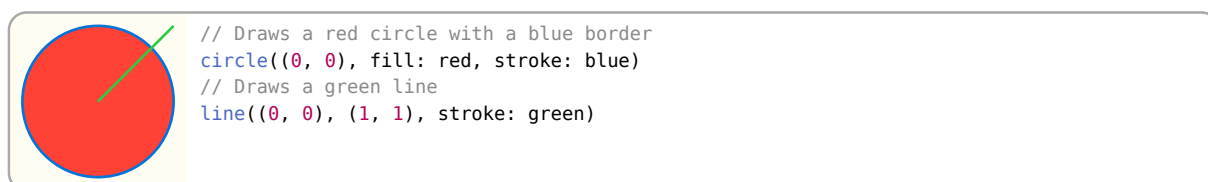
Default: none

How to fill the drawn element.

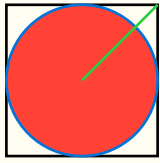
stroke: none or auto or length or color or dictionary or stroke

Default: 1pt + luma(0%)

How to stroke the border or the path of the draw element. See Typst's line documentation for more details: <https://typst.app/docs/reference/visualize/line/#parameters-stroke>



Instead of having to specify the same styling for each time you want to draw an element, you can use the `set-style()` function to change the style for all elements after it. You can still pass styling to a draw function to override what has been set with `set-style()`. You can also use the `fill()` and `stroke()` functions as a shorthand to set the fill and stroke respectively.



```
// Draws an empty square with a black border
rect((-1, -1), (1, 1))

// Sets the global style to have a fill of red and a stroke of blue
set-style(stroke: blue, fill: red)
circle((0,0))

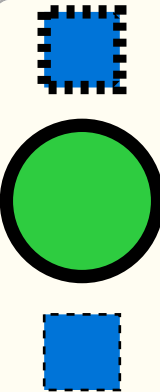
// Draws a green line despite the global stroke is blue
line((0, (1,1), stroke: green)
```

When using a dictionary for a style, it is important to note that they update each other instead of overriding the entire option like a non-dictionary value would do. For example, if the stroke is set to (paint: red, thickness: 5pt) and you pass (paint: blue), the stroke would become (paint: blue, thickness: 5pt).



```
// Sets the stroke to red with a thickness of 5pt
set-style(stroke: (paint: red, thickness: 5pt))
// Draws a line with the global stroke
line((0,0), (1,0))
// Draws a blue line with a thickness of 5pt because dictionaries update the style
line((0,0), (1,1), stroke: (paint: blue))
// Draws a yellow line with a thickness of 1pt because other values override the style
line((0,0), (0,1), stroke: yellow)
```

You can also specify styling for each type of element. Note that dictionary values will still update with its global value, the full hierarchy is function > element type > global. When the value of a style is auto, it will become exactly its parent style.



```
set-style(
  // Global fill and stroke
  fill: green,
  stroke: (thickness: 5pt),
  // Stroke and fill for only rectangles
  rect: (stroke: (dash: "dashed"), fill: blue),
)
rect((0,0), (1,1))
circle((0.5, -1.5))
rect((0,-3), (1, -4), stroke: (thickness: 1pt))
```



```
// Its a nice drawing okay
set-style(
  rect: (
    fill: red,
    stroke: none
  ),
  line: (
    fill: blue,
    stroke: (dash: "dashed")
  ),
)
rect((0,0), (1,1))
line((0, -1.5), (0.5, -0.5), (1, -1.5), close: true)
circle((0.5, -2.5), radius: 0.5, fill: green)
```

3.2.1 Marks

Marks are arrow tips that can be added to the end of path based elements that support the mark style key, or can be directly drawn by using the mark draw function. Marks are specified by giving there

names as strings and have several options to customise them. You can give an array of names to have multiple marks in a row, dictionaries can also be used in the array for per mark styling.

Name	Shorthand	Shape
triangle	>, < (reversed)	—▶
stealth		—▶
bar		—
ellipse		—○
circle	o	—○
bracket], [(reversed)	—⌋
diamond		—◇
rect	[]	—□
hook		—⌋
straight		—▶
barbed		—▶
plus	+	—+
x	x	—×
star	*	—*

Table 1: Mark symbols

```
let c = ((rel: (0, -1)), (rel: (2, 0), update: false)) // Coordinates to draw the line, it
is not necessary to understand this for this example.
// No marks
line(), (rel: (1, 0), update: false))
// Draws a triangle mark at both ends of the line.
set-style(mark: (symbol: ">"))
line(..c)
// Overrides the end mark to be a diamond but the start is still a triangle.
set-style(mark: (end: "<>"))
line(..c)
// Draws two triangle marks at both ends but the first mark of end is still a diamond.
set-style(mark: (symbol: (">", ">")))
line(..c)
// Sets the stroke of first mark in the sequence to red but the end mark overrides it to
be blue.
set-style(mark: (symbol: ((symbol: ">", stroke: red), ">"), end: (stroke: blue)))
line(..c)
```

symbol: `none` or `string` or `array` or `dictionary`

Default: `none`

This option sets the mark to draw when using the mark draw function, or applies styling to both mark ends of path based elements. The mark's name or shorthand can be given, multiple marks can be drawn by passing an array of names or shorthands. When `none` no marks will be drawn. A style dictionary can be given instead of a `string` to override styling for that particular mark, just make sure to still give the mark name using the `symbol` key otherwise nothing will be drawn!

start: `none` or `string` or `array` or `dictionary`

Default: `none`

This option sets the mark to draw at the start of a path based element. It will override all options of the `symbol` key and will not effect marks drawn using the mark draw function.

end: `none` or `string` or `array` or `dictionary`

Default: `none`

This option sets the mark to draw at the end of a path based element. It will override all options of the `symbol` key and will not effect marks drawn using the mark draw function.

length: number	Default: 5.67pt
The size of the mark in the direction it is pointing.	
width: number	Default: 4.25pt
The size of the mark along the normal of its direction.	
inset: number	Default: 1.42pt
It specifies a distance by which something inside the arrow tip is set inwards; for the Stealth arrow tip it is the distance by which the back angle is moved inwards.	
scale: float	Default: 1
A factor that is applied to the mark's length, width and inset.	
sep: number	Default: 2.83pt
The distance between multiple marks along their path.	
flex: boolean	Default: true
Only applicable when marks are used on curves such as bezier and hobby. If true, the mark will point along the secant of the curve. If false, the tangent at the marks tip is used.	
position-samples: integer	Default: 30
Only applicable when marks are used on curves such as bezier and hobby. The maximum number of samples to use for calculating curve positions. A higher number gives better results but may slow down compilation.	
pos: number or ratio	Default: none
Overrides the mark's position along a path. A number will move it an absolute distance, while a ratio will be a distance relative to the length of the path. Note that this may be removed in the future in preference of a different method.	
offset: number or ratio	Default: none
Like pos but it moves the position of the mark instead of overriding it.	
anchor: string	Default: "tip"
Anchor of the mark to use for positioning. Available anchors are	
<ul style="list-style-type: none"> • tip The marks tip (default) • center The visual center of the mark • base The base/end of the mark 	
slant: ratio	Default: 0%
How much to slant the mark relative to the axis of the arrow. 0% means no slant 100% slants at 45 degrees	
harpoon: boolean	Default: false
When true only the top half of the mark is drawn.	
flip: boolean	Default: false
When true the mark is flipped along its axis.	
reverse: boolean	Default: false
Reverses the direction of the mark.	
xy-up: vector	Default: (0, 0, 1)

The direction which is “up” for use when drawing 2D marks.

z-up: vector

Default: (0, 1, 0)

The direction which is “up” for use when drawing 3D marks.

shorten-to: integer or auto or none

Default: auto

Which mark to shorten the path to when multiple marks are given. auto will shorten to the last mark, none will shorten to the first mark (effectively disabling path shortening). An integer can be given to select the mark’s index.

transform-shape: bool

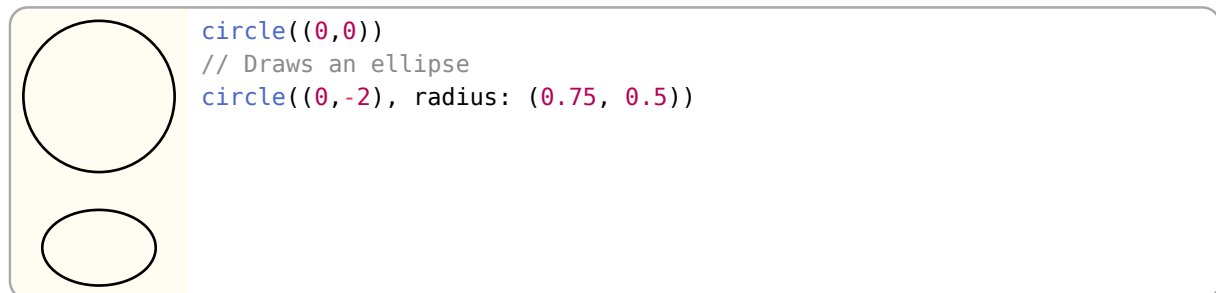
Default: true

When false marks will not be stretched/affected by the current transformation, marks will be placed after the path is transformed.

3.3 Shapes

3.3.1 circle

Draws a circle or ellipse.



Parameters

```
circle(
  position: coordinate,
  name: none string,
  anchor: none string,
  ..style: style
)
```

position: coordinate

The position to place the circle on.

Styling

Root: circle

Keys

radius: number or array

Default: 1

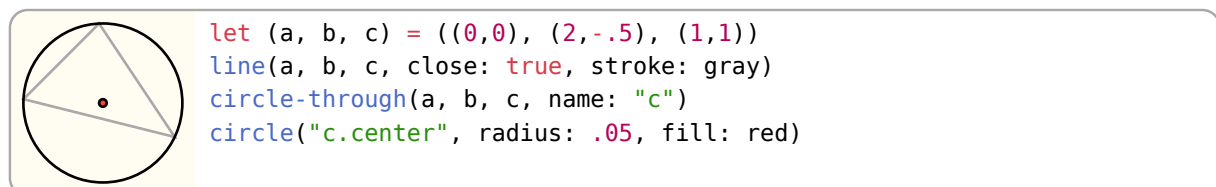
A number that defines the size of the circle's radius. Can also be set to a tuple of two numbers to define the radii of an ellipse, the first number is the x radius and the second is the y radius.

Anchors

Supports border and path anchors. The “center” anchor is the default.

3.3.2 circle-through

Draws a circle through three coordinates.



Parameters

```
circle-through(
  a: coordinate,
  b: coordinate,
  c: coordinate,
  name: none string,
  anchor: none string,
  ..style: style
)
```

a: coordinate

Coordinate a.

b: coordinate

Coordinate b.

c: coordinate

Coordinate c.

Styling

Root: circle

circle-through has the same styling as `circle()` except for radius as the circle's radius is calculated by the given coordinates.

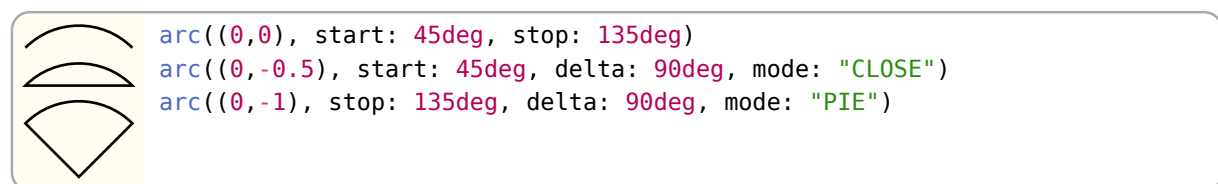
Anchors

Supports the same anchors as `circle` as well as:

- a** Coordinate a
- b** Coordinate b
- c** Coordinate c

3.3.3 arc

Draws a circular segment.



Note that two of the three angle arguments (start, stop and delta) must be set. The current position `()` gets updated to the arc's end coordinate (anchor `arc-end`).

Parameters

```
arc(
  position: coordinate,
  start: auto angle,
  stop: auto angle,
  delta: auto angle,
  name: none string,
  anchor: none string,
  ..style: style
)
```

position: `coordinate`

Position to place the arc at.

start: `auto` or `angle`Default: `auto`

The angle at which the arc should start. Remember that `0deg` points directly towards the right and `90deg` points up.

stop: `auto` or `angle`Default: `auto`

The angle at which the arc should stop.

delta: `auto` or `angle`Default: `auto`

The change in angle away start or stop.

Styling

Root: `arc`

Keys

radius: `number` or `array`Default: `1`

The radius of the arc. An elliptical arc can be created by passing a tuple of numbers where the first element is the x radius and the second element is the y radius.

mode: `string`Default: `"OPEN"`

The options are: "OPEN" no additional lines are drawn so just the arc is shown; "CLOSE" a line is drawn from the start to the end of the arc creating a circular segment; "PIE" lines are drawn from the start and end of the arc to the origin creating a circular sector.

update-position: `bool`Default: `true`

Update the current canvas position to the arc's end point (anchor "arc-end"). This overrides the default of `true`, that allows chaining of (arc) elements.

Anchors

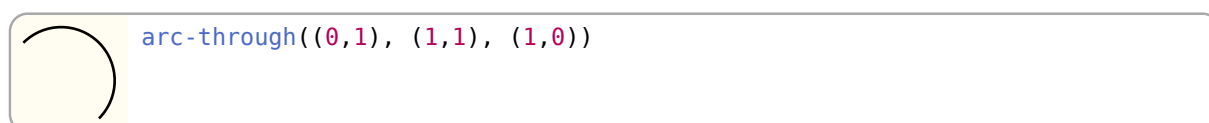
Supports border and path anchors.

arc-start The position at which the arc's curve starts, this is the default.**arc-end** The position of the arc's curve end.**arc-center** The midpoint of the arc's curve.**center** The center of the arc, this position changes depending on if the arc is closed or not.**chord-center** Center of chord of the arc drawn between the start and end point.**origin** The origin of the arc's circle.

3.3.4 arc-through

Draws an arc that passes through three points a, b and c.

Note that all three points must not lie on a straight line, otherwise the function fails.

`arc-through((0,1), (1,1), (1,0))`

Parameters

```
arc-through(
  a: coordinate,
  b: coordinate,
  c: coordinate,
  name: none string,
  ..style: style
)
```

a: coordinate

Start position of the arc

b: coordinate

Position the arc passes through

c: coordinate

End position of the arc

Styling

Root: arc

Uses the same styling as `arc()`

Anchors

For anchors see `arc()`.

3.3.5 mark

Draws a single mark pointing towards a target coordinate.



```
mark((0,0), (1,0), symbol: ">", fill: black)
mark((0,0), (1,1), symbol: "stealth", scale: 3, fill: black)
```

Note: To place a mark centered at the first coordinate (from) use the marks anchor: "center" style.

Parameters

```
mark(
  from: coordinate,
  to: coordinate angle,
  ..style: style
)
```

from: coordinate

The position to place the mark.

to: coordinate or angle

The position or angle the mark should point towards.

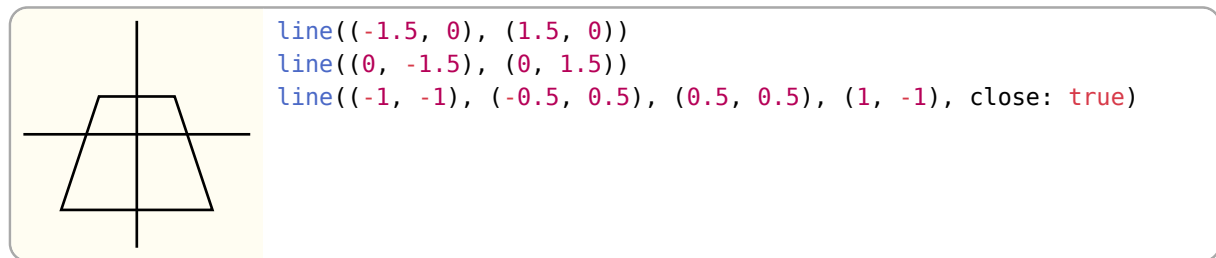
Styling

Root: mark

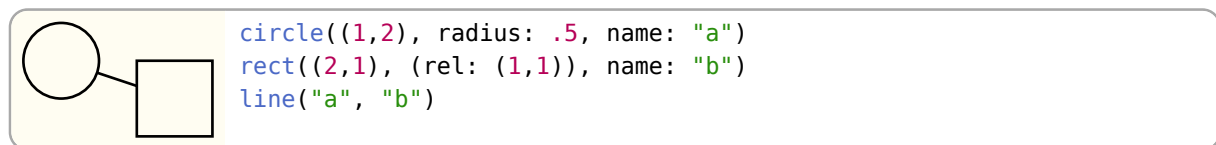
You can directly use the styling from Section 3.2.1.

3.3.6 line

Draws a line, more than two points can be given to create a line-strip.



If the first or last coordinates are given as the name of an element, that has a "default" anchor, the intersection of that element's border and a line from the first or last two coordinates given is used as coordinate. This is useful to span a line between the borders of two elements.



Parameters

```
line(
  ..pts-style: coordinates style,
  close: bool,
  name: none string
)
```

..pts-style: coordinates or style

Positional two or more coordinates to draw lines between. Accepts style key-value pairs.

close: bool

Default: false

If true, the line-strip gets closed to form a polygon

Styling

Root: line

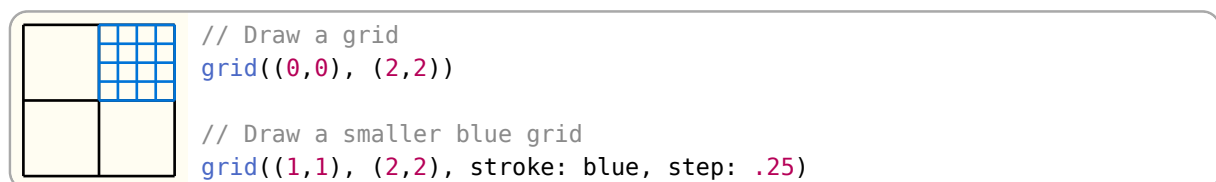
Supports mark styling.

Anchors

Supports path anchors.

3.3.7 grid

Draws a grid between two coordinates



Parameters

```
grid(
  from: coordinate,
  to: coordinate,
  name: none string,
  ..style: style
)
```

from: coordinate

The top left of the grid

to: coordinate

The bottom right of the grid

Styling

Root: grid

Keys

step: number or tuple or dictionary

Default: 1

Distance between grid lines. A distance of 1 means to draw a grid line every 1 length units in x- and y-direction. If given a dictionary with x and y keys or a tuple, the step is set per axis.

help-lines: bool

Default: 1

If true, force the stroke style to gray + 0.2pt

Anchors

Supports border anchors.

3.3.8 content

Positions Typst content in the canvas. Note that the content itself is not transformed only its position is.

Hello World! `content((0,0), [Hello World!])`

To put text on a line you can let the function calculate the angle between its position and a second coordinate by passing it to angle:

Text on a line

```
line((0, 0), (3, 1), name: "line")
content(
  ("line.start", 0.5, "line.end"),
  angle: "line.end",
  padding: .1,
  anchor: "south",
  [Text on a line]
)
```

This is
a long
text.

```
// Place content in a rect between two coordinates
content((0, 0), (2, 2), box(par(justify: false)[This is a long text.],
stroke: 1pt, width: 100%, height: 100%, inset: 1em))
```

Parameters

```
content(
  ..args-style: coordinate content style,
  angle: angle coordinate,
  anchor: none string,
  name: none string
)
```

..args-style: coordinate or content or style

When one coordinate is given as a positional argument, the content will be placed at that position. When two coordinates are given as positional arguments, the content will be placed inside a rectangle between the two positions. All named arguments are styling and any additional positional arguments will panic.

angle: angle or coordinate

Default: 0deg

Rotates the content by the given angle. A coordinate can be given to rotate the content by the angle between it and the first coordinate given in args. This effectively points the right hand side of the content towards the coordinate. This currently exists because Typst's rotate function does not change the width and height of content.

Styling

Root: content

Keys

padding: number or dictionary

Default: 0

Sets the spacing around content. Can be a single number to set padding on all sides or a dictionary to specify each side specifically. The dictionary follows Typst's pad function: <https://typst.app/docs/reference/layout/pad/>

frame: string or none

Default: none

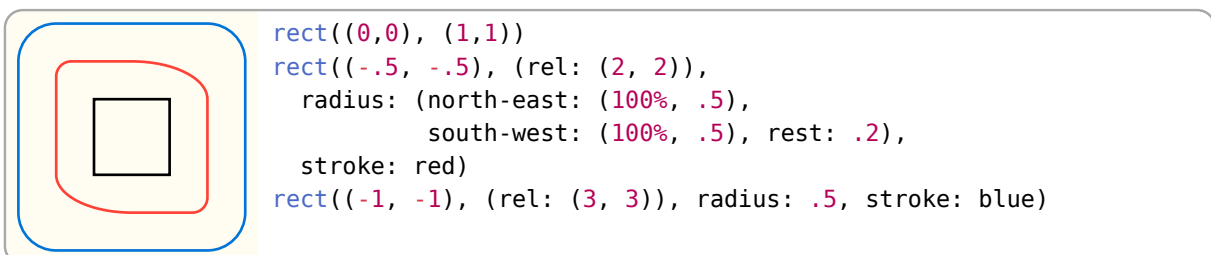
Sets the frame style. Can be none, "rect" or "circle" and inherits the stroke and fill style.

Anchors

Supports border anchors.

3.3.9 rect

Draws a rectangle between two coordinates.



Parameters

```
rect(
  a: coordinate,
  b: coordinate,
  name: none string,
  anchor: none string,
  ..style: style
)
```

a: coordinate

Coordinate of the bottom left corner of the rectangle.

b: coordinate

Coordinate of the top right corner of the rectangle. You can draw a rectangle with a specified width and height by using relative coordinates for this parameter (`rel: (width, height)`).

Styling

Root rect

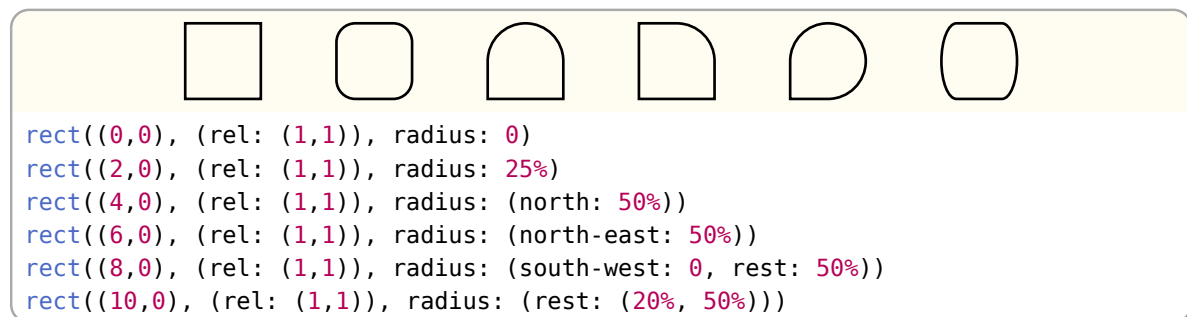
Keys

radius: number or ratio or dictionary

Default: 0

The rectangles corner radius. If set to a single number, that radius is applied to all four corners of the rectangle. If passed a dictionary you can set the radii per corner. The following keys support either a number, ratio or an array of number, ratio for specifying a different x- and y-radius: north, east, south, west, north-west, north-east, south-west and south-east. To set a default value for remaining corners, the `rest` key can be used.

Ratio values are relative to the rects width/height.

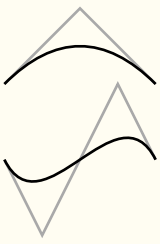


Anchors

Supports border and path anchors.

3.3.10 bezier

Draws a quadratic or cubic bezier curve



```
let (a, b, c) = ((0, 0), (2, 0), (1, 1))
line(a, c, b, stroke: gray)
bezier(a, b, c)

let (a, b, c, d) = ((0, -1), (2, -1), (.5, -2), (1.5, 0))
line(a, c, d, b, stroke: gray)
bezier(a, b, c, d)
```

Parameters

```
bezier(
  start: coordinate,
  end: coordinate,
  ..ctrl-style: coordinate style,
  name: none string
)
```

start: coordinate

Start position

end: coordinate

End position (last coordinate)

..ctrl-style: coordinate or style

The first two positional arguments are taken as cubic bezier control points, where the first is the start control point and the second is the end control point. One control point can be given for a quadratic bezier curve instead. Named arguments are for styling.

Styling

Root bezier

Supports marks.

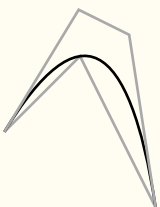
Anchors

Supports path anchors.

ctrl-n nth control point where n is an integer starting at 0

3.3.11 bezier-through

Draws a cubic bezier curve through a set of three points. See bezier for style and anchor details.



```
let (a, b, c) = ((0, 0), (1, 1), (2, -1))
line(a, b, c, stroke: gray)
bezier-through(a, b, c, name: "b")

// Show calculated control points
line(a, "b.ctrl-0", "b.ctrl-1", c, stroke: gray)
```

Parameters

```
bezier-through(
  start: coordinate,
  pass-through: coordinate,
  end: coordinate,
  name: none string,
  ..style: style
)
```

start: coordinate

The position to start the curve.

pass-through: coordinate

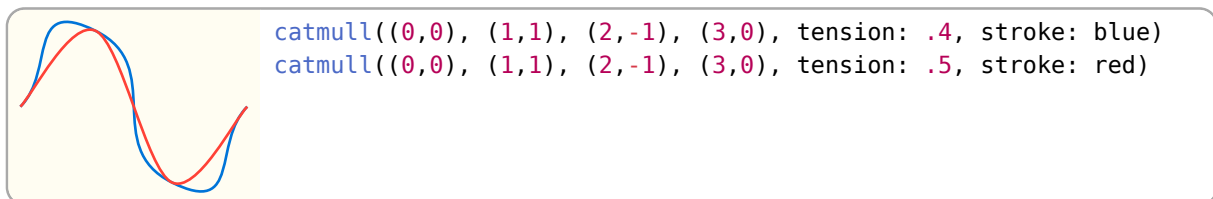
The position to pass the curve through.

end: coordinate

The position to end the curve.

3.3.12 catmull

Draws a Catmull-Rom curve through a set of points.



Parameters

```
catmull(
  ..pts-style: coordinate style,
  close: bool,
  name: none string
)
```

..pts-style: coordinate or style

Positional arguments should be coordinates that the curve should pass through. Named arguments are for styling.

close: bool

Default: **false**

Closes the curve with a straight line between the start and end of the curve.

Styling

Root catmull

Supports marks.

Keys

tension: float

Default: **0.5**

How tight the curve should fit to the points. The higher the tension the less curvy the curve.

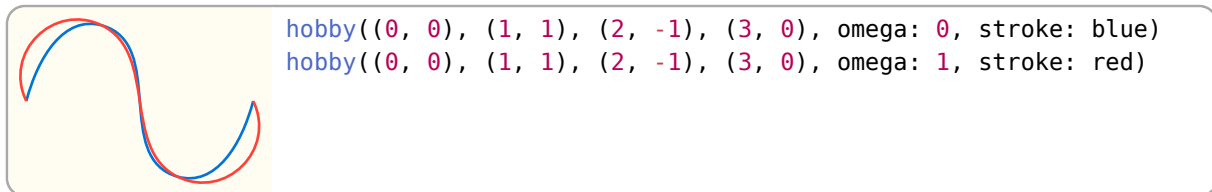
Anchors

Supports path anchors.

pt-n The nth given position (0 indexed so “pt-0” is equal to “start”)

3.3.13 hobby

Draws a Hobby curve through a set of points.



Parameters

```
hobby(
  ..pts-style: coordinate style,
  ta: auto array,
  tb: auto array,
  close: bool,
  name: none string
)
```

..pts-style: coordinate or style

Positional arguments are the coordinates to use to draw the curve with, a minimum of two is required. Named arguments are for styling.

ta: auto or array

Default: auto

Outgoing tension at pts.at(n) from pts.at(n) to pts.at(n+1). The number given must be one less than the number of points.

tb: auto or array

Default: auto

Incoming tension at pts.at(n+1) from pts.at(n) to pts.at(n+1). The number given must be one less than the number of points.

close: bool

Default: false

Closes the curve with a proper smooth curve between the start and end of the curve.

Styling

Root hobby

Supports marks.

Keys

omega: tuple of float

Default: (1, 1)

How curly the curve should be at each endpoint. When the curl is close to zero, the spline approaches a straight line near the endpoints. When the curl is close to one, it approaches a circular arc.

Aliases

Supports path anchors.

pt-n The nth given position (0 indexed, so “pt-0” is equal to “start”)

3.3.14 merge-path

Merges two or more paths by concatenating their elements. Anchors and visual styling, such as `stroke` and `fill`, are not preserved. When an element's path does not start at the same position the previous element's path ended, a straight line is drawn between them so that the final path is continuous. You must then pay attention to the direction in which element paths are drawn.



```
merge-path(fill: white, {
  line((0, 0), (1, 0))
  bezier((0, 0), (1,1), (0,1))
})
```

Elements hidden via `hide()` are ignored.

Parameters

```
merge-path(
  body: elements,
  close: bool,
  name: none string,
  ..style: style
)
```

body: `elements`

Elements with paths to be merged together.

close: `bool`

Default: `false`

Close the path with a straight line from the start of the path to its end.

Anchors

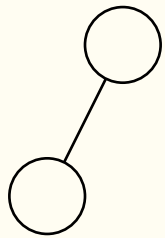
Supports path anchors.

3.4 Grouping

3.4.1 hide

Hides an element.

Hidden elements are not drawn to the canvas, are ignored when calculating bounding boxes and discarded by merge-path. All other behaviours remain the same as a non-hidden element.



```
set-style(radius: .5)
intersections("i", {
  circle((0,0), name: "a")
  circle((1,2), name: "b")
  // Use a hidden line to find the border intersections
  hide(line("a.center", "b.center"))
})
line("i.0", "i.1")
```

Parameters

```
hide(
  body: element,
  bounds: bool
)
```

body: element

One or more elements to hide

bounds: bool

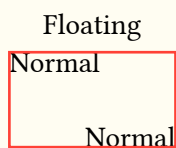
Default: **false**

If true, respect the bounding box of the hidden elements for resizing the canvas

3.4.2 floating

Places an element without affecting bounding boxes.

Floating elements are drawn to the canvas but are ignored when calculating bounding boxes. All other behaviours remain the same.



```
group(name: "g", {
  content((1,0), [Normal])
  content((0,1), [Normal])
  floating(content((.5,1.5), [Floating]))
})
set-style(stroke: red)
rect("g.north-west", "g.south-east")
```

Parameters

```
floating(body: element)
```

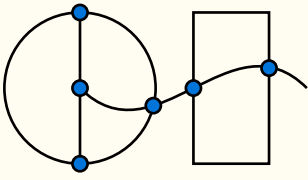
body: element

One or more elements to place

3.4.3 intersections

Calculates the intersections between multiple paths and creates one anchor per intersection point.

All resulting anchors will be named numerically, starting at 0. i.e., a call `intersections("a", ...)` will generate the anchors "a.0", "a.1", "a.2" to "a.n", depending of the number of intersections.

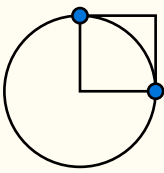


```

intersections("i", {
  circle((0, 0))
  bezier((0,0), (3,0), (1,-1), (2,1))
  line((0,-1), (0,1))
  rect((1.5,-1),(2.5,1))
})
for-each-anchor("i", (name) => {
  circle("i." + name, radius: .1, fill: blue)
})

```

You can also use named elements:



```

circle((0,0), name: "a")
rect((0,0), (1,1), name: "b")
intersections("i", "a", "b")
for-each-anchor("i", (name) => {
  circle("i." + name, radius: .1, fill: blue)
})

```

You can calculate intersections with hidden elements by using `hide()`.

Parameters

```

intersections(
  name: string,
  ..elements: elements string,
  samples: int
)

```

name: string

Name to prepend to the generated anchors. (Not to be confused with other name arguments that allow the use of anchor coordinates.)

..elements: elements or string

Elements and/or element names to calculate intersections with. Elements referred to by name are (unlike elements passed) not drawn by the intersections function!

samples: int

Default: 10

Number of samples to use for non-linear path segments. A higher sample count can give more precise results but worse performance.

3.4.4 group

Groups one or more elements together. This element acts as a scope, all state changes such as transformations and styling only affect the elements in the group. Elements after the group are not affected by the changes inside the group.



```
// Create group
group({
  stroke(5pt)
  scale(.5); rotate(45deg)
  rect((-1,-1),(1,1))
})
rect((-1,-1),(1,1))
```

Parameters

```
group(
  body: elements function,
  name: none string,
  anchor: none string,
  ..style: style
)
```

body: elements or function

Elements to group together. A least one is required. A function that accepts `ctx` and returns elements is also accepted.

anchor: none or string

Default: none

Anchor to position the group and it's children relative to. For translation the difference between the groups "default" anchor and the passed anchor is used.

Styling

Root group

Keys

padding: none or number or array or dictionary

Default: none

How much padding to add around the group's bounding box. none applies no padding. A number applies padding to all sides equally. A dictionary applies padding following Typst's `pad` function: <https://typst.app/docs/reference/layout/pad/>. An array follows CSS like padding: (y, x), (top, x, bottom) or (top, right, bottom, left).

Anchors

Supports border and path anchors. However they are created based on the axis aligned bounding box of all the child elements of the group.

You can add custom anchors to the group by using the `anchor` element while in the scope of said group, see `anchor` for more details. You can also copy over anchors from named child element by using the `copy-anchors` element as they are not accessible from outside the group.

The default anchor is "center" but this can be overridden by using `anchor` to place a new anchor called "default".

Named elements within a group can also be accessed as string anchors, see Section 4.6.

3.4.5 scope

This element acts as a scope, all state changes such as transformations and styling only affect the elements in the group. Elements after the scope are not affected by the changes inside the scope. In

contrast to `group`, the `scope` element does not create a named element itself and “leaks” body element to the outside.

Parameters

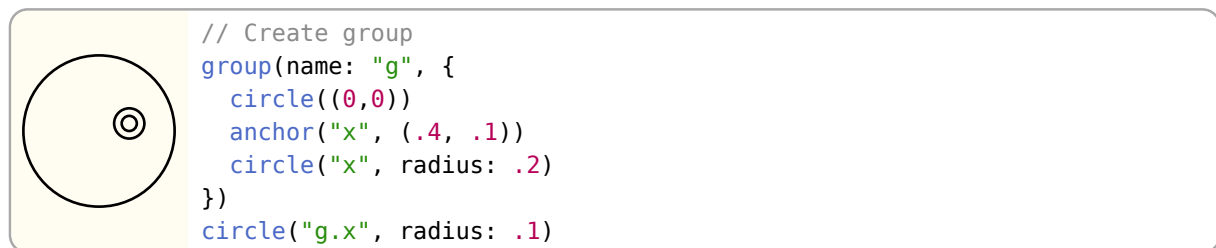
`scope`(body: `elements` `function`)

body: `elements` or `function`

Elements to group together. A least one is required. A function that accepts `ctx` and returns elements is also accepted.

3.4.6 anchor

Creates a new anchor for the current group. This element can only be used inside a group otherwise it will panic. The new anchor will be accessible from inside the group by using just the anchor’s name as a coordinate.



Parameters

`anchor`(
 name: `string`,
 position: `coordinate`
)

name: `string`

The name of the anchor

position: `coordinate`

The position of the anchor

3.4.7 copy-anchors

Copies multiple anchors from one element into the current group. Panics when used outside of a group. Copied anchors will be accessible in the same way anchors created by the `anchor` element are.

Parameters

`copy-anchors`(
 element: `string`,
 filter: `auto` `array`
)

element: `string`

The name of the element to copy anchors from.

filter: `auto` or `array`

Default: `auto`

When set to `auto` all anchors will be copied to the group. An array of anchor names can instead be given so only the anchors that are in the element and the list will be copied over.

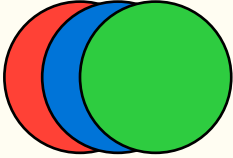
3.4.8 set-ctx

An advanced element that allows you to modify the current canvas context.

A context object holds the canvas' state, such as the element dictionary, the current transformation matrix, group and canvas unit length. The following fields are considered stable:

- `length` (`length`): Length of one canvas unit as typst length
- `transform` (`cetz.matrix`): Current 4x4 transformation matrix
- `debug` (`bool`): True if the canvas' debug flag is set

Note: The transformation matrix (`transform`) is rounded after calling the `callback` function and therefore might be not exactly the matrix specified. This is due to rounding errors and should not cause any problems.



```
// Setting a custom transformation matrix
set-ctx(ctx => {
  let mat = ((1, 0, .5, 0),
             (0, 1, 0, 0),
             (0, 0, 1, 0),
             (0, 0, 0, 1))

  ctx.transform = mat
  return ctx
})
circle((z: 0), fill: red)
circle((z: 1), fill: blue)
circle((z: 2), fill: green)
```

Parameters

`set-ctx`(callback: `function`)

callback: `function`

A function that accepts the context dictionary and only returns a new one.

3.4.9 get-ctx

An advanced element that allows you to read the current canvas context through a callback and return elements based on it.

```
(
  (1, 0, -0.5, 0),
  (0, -1, 0.5, 0),
  (0, 0, 0.0, 0),
  (0, 0, 0.0, 1),
)
```

```
// Print the transformation matrix
get-ctx(ctx => {
  content(), [#repr(ctx.transform)]
})
```

Parameters

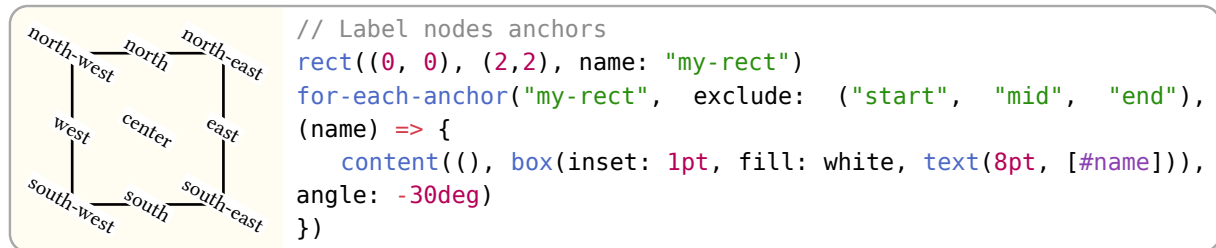
`get-ctx`(callback: `function`)

callback: `function`

A function that accepts the context dictionary and can return elements.

3.4.10 for-each-anchor

Iterates through all named anchors of an element and calls a callback for each one.



Parameters

```
for-each-anchor(
  name: string,
  callback: function,
  exclude: array
)
```

name: string

The name of the element with the anchors to loop through.

callback: function

A function that takes the anchor name and can return elements.

exclude: array

Default: ()

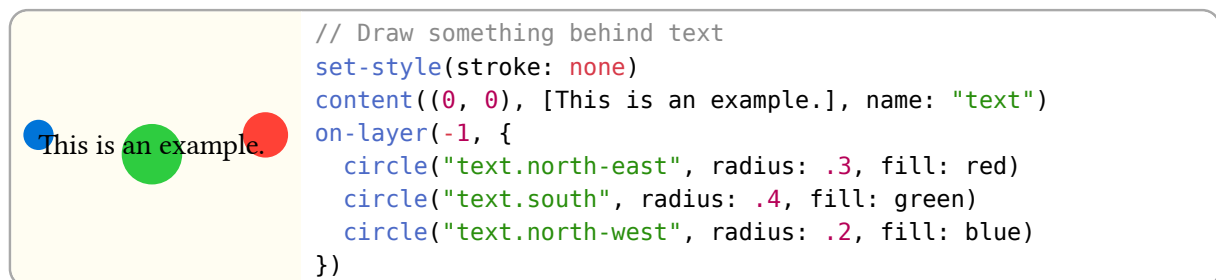
An array of anchor names to not include in the loop.

3.4.11 on-layer

Places elements on a specific layer.

A layer determines the position of an element in the draw queue. A lower layer is drawn before a higher layer.

Layers can be used to draw behind or in front of other elements, even if the other elements were created before or after. An example would be drawing a background behind a text, but using the text's calculated bounding box for positioning the background.



Parameters

```
on-layer(  
  layer: float integer,  
  body: elements function  
)
```

layer: float or integer

The layer to place the elements on. Elements placed without on-layer are always placed on layer 0.

body: elements or function

Elements to draw on the layer specified. A function that accepts ctx and returns elements is also accepted.

3.5 Utility

3.5.1 assert-version

Assert that the cetz version of the canvas matches the given version (range).

min (version): Minimum version (current \geq min) max (none, version): First unsupported version (current $<$ max) hint (string): Name of the function/module this assert is called from

Parameters

```
assert-version(  
    min,  
    max,  
    hint  
)
```

min:

max:

Default: **none**

hint:

Default: **" "**

3.6 Transformations

All transformation functions push a transformation matrix onto the current transform stack. To apply transformations scoped use a `group(...)` object.

Transformation matrices get multiplied in the following order:

$$M_{\text{world}} = M_{\text{world}} \cdot M_{\text{local}}$$

3.6.1 set-transform

Sets the transformation matrix.

Parameters

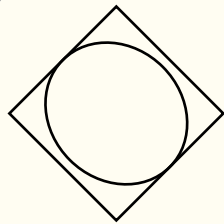
`set-transform(mat: none matrix)`

mat: `none` or `matrix`

The 4x4 transformation matrix to set. If `none` is passed, the transformation matrix is set to the identity matrix (`matrix.ident()`).

3.6.2 rotate

Rotates the transformation matrix on the z-axis by a given angle or other axes when specified.



```
// Rotate on z-axis
rotate(z: 45deg)
rect((-1, -1), (1, 1))
// Rotate on y-axis
rotate(y: 80deg)
circle((0, 0))
```

Parameters

```
rotate(
  ..angles: angle,
  origin: none coordinate
)
```

..angles: `angle`

A single angle as a positional argument to rotate on the z-axis by. Named arguments of `x`, `y` or `z` can be given to rotate on their respective axis. You can give named arguments of `yaw`, `pitch` or `roll`, too.

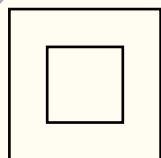
origin: `none` or `coordinate`

Default: `none`

Origin to rotate around, or `(0, 0, 0)` if set to `none`.

3.6.3 translate

Translates the transformation matrix by the given vector or dictionary.



```
// Outer rect
rect((0, 0), (2, 2))
// Inner rect
translate(x: .5, y: .5)
rect((0, 0), (1, 1))
```

Parameters

```
translate(
  ..args: vector float length,
  pre: bool
)
```

..args: vector or float or length

A single vector or any combination of the named arguments x, y and z to translate by. A translation matrix with the given offsets gets multiplied with the current transformation depending on the value of pre.

pre: bool

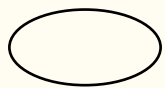
Default: **false**

Specify matrix multiplication order

- false: World = World * Translate
- true: World = Translate * World

3.6.4 scale

Scales the transformation matrix by the given factor(s).



```
// Scale the y-axis
scale(y: 50%)
circle((0,0))
```

Parameters

```
scale(
  ..args: float ratio,
  origin: none coordinate
)
```

..args: float or ratio

A single value to scale the transformation matrix by or per axis scaling factors. Accepts a single float or ratio value or any combination of the named arguments x, y and z to set per axis scaling factors. A ratio of 100% is the same as the value 1.

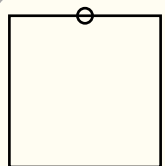
origin: none or coordinate

Default: **none**

Origin to rotate around, or (0, 0, 0) if set to none.

3.6.5 set-origin

Sets the given position as the new origin (0, 0, 0)



```
// Outer rect
rect((0,0), (2,2), name: "r")
// Move origin to top edge
set-origin("r.north")
circle((0, 0), radius: .1)
```

Parameters

```
set-origin(origin: coordinate)
```

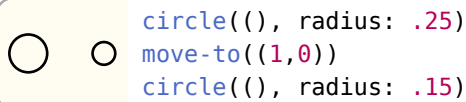
origin: coordinate

Coordinate to set as new origin (0,0,0)

3.6.6 move-to

Sets the previous coordinate.

The previous coordinate can be used via `()` (empty coordinate). It is also used as base for relative coordinates if not specified otherwise.



```
circle(), radius: .25
move-to((1,0))
circle(), radius: .15
```

Parameters

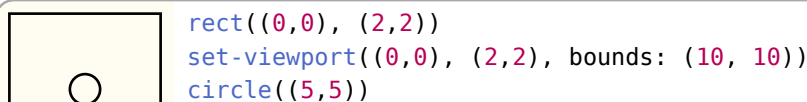
`move-to`(pt: coordinate)

pt: coordinate

The coordinate to move to.

3.6.7 set-viewport

Span viewport between two coordinates and set-up scaling and translation



```
rect((0,0), (2,2))
set-viewport((0,0), (2,2), bounds: (10, 10))
circle((5,5))
```

Parameters

```
set-viewport(
  from: coordinate,
  to: coordinate,
  bounds: vector
)
```

from: coordinate

Bottom left corner coordinate

to: coordinate

Top right corner coordinate

bounds: vector

Default: (1, 1, 1)

Viewport bounds vector that describes the inner width, height and depth of the viewport

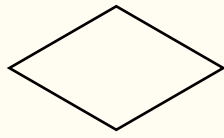
3.7 Projection

3.7.1 ortho

Set-up an orthographic projection environment.

This is a transformation matrix that rotates elements around the x, the y and the z axis by the parameters given.

By default an isometric projection ($x \approx 35.264^\circ$, $y = 45^\circ$) is set.



```
ortho({
  on-xz({
    rect((-1,-1), (1,1))
  })
})
```

Parameters

```
ortho(
  x: angle,
  y: angle,
  z: angle,
  sorted: bool,
  cull-face: none string,
  reset-transform: bool,
  body: element,
  name
)
```

x: `angle` Default: `35.264deg`

X-axis rotation angle

y: `angle` Default: `45deg`

Y-axis rotation angle

z: `angle` Default: `0deg`

Z-axis rotation angle

sorted: `bool` Default: `true`

Sort drawables by maximum distance (front to back)

cull-face: `none` or `string` Default: `none`

Enable back-face culling if set to "cw" for clockwise or "ccw" for counter-clockwise. Polygons of the specified order will not get drawn.

reset-transform: `bool` Default: `false`

Ignore the current transformation matrix

body: `element`

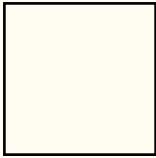
Elements to draw

name: Default: `none`

3.7.2 on-xy

Draw elements on the xy-plane with optional z offset.

All vertices of all elements will be changed in the following way: $\begin{pmatrix} x \\ y \\ z_{\text{argument}} \end{pmatrix}$, where z_{argument} is the z-value given as argument.



```
on-xy({
  rect((-1, -1), (1, 1))
})
```

Parameters

```
on-xy(
  z: number,
  body: element
)
```

z: number

Default: 0

Z offset for all coordinates

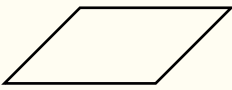
body: element

Elements to draw

3.7.3 on-xz

Draw elements on the xz-plane with optional y offset.

All vertices of all elements will be changed in the following way: $\begin{pmatrix} x \\ y_{\text{argument}} \\ y \end{pmatrix}$, where y_{argument} is the y-value given as argument.



```
on-xz({
  rect((-1, -1), (1, 1))
})
```

Parameters

```
on-xz(
  y: number,
  body: element
)
```

y: number

Default: 0

Y offset for all coordinates

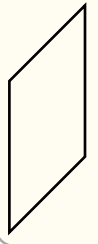
body: element

Elements to draw

3.7.4 on-yz

Draw elements on the yz-plane with optional x offset.

All vertices of all elements will be changed in the following way: $\begin{pmatrix} x_{\text{argument}} \\ x \\ y \end{pmatrix}$, where x_{argument} is the x-value given as argument.



```
on-yz({
  rect((-1, -1), (1, 1))
})
```

Parameters

```
on-yz(
  x: number,
  body: element
)
```

x: number

Default: 0

X offset for all coordinates

body: element

Elements to draw

3.8 Bodies

3.8.1 prism

Draw a prism by extending a single element into a direction.

Curved shapes get sampled into linear ones.

Parameters

```
prism(
  front-face: elements,
  dir: number vector,
  samples: int,
  ..style
)
```

front-face: elements

A single element to use as front-face

dir: number or vector

Z-distance or direction vector to extend the front-face along

samples: int

Default: 10

Number of samples to use for sampling curves

..style:

Styling

Root: prism

Keys

front-stroke: stroke or none

Default: auto

Front-face stroke

front-fill: fill or none

Default: auto

Front-face fill

back-stroke: stroke or none

Default: auto

Back-face stroke

back-fill: fill or none

Default: auto

Back-face fill

side-stroke: stroke or none

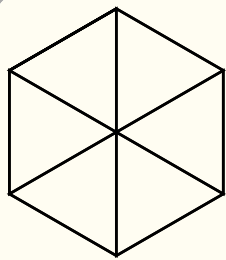
Default: auto

Side stroke

side-fill: fill or none

Default: auto

Side fill



```
ortho({
  // Draw a cube with and edge length of 2
  prism({
    rect((-1, -1), (rel: (2, 2)))
  }, 2)
})
```

4 Coordinate Systems

A *coordinate* is a position on the canvas on which the picture is drawn. They take the form of dictionaries and the following sub-sections define the key value pairs for each system. Some systems have a more implicit form as an array of values and CeTZ attempts to infer the system based on the element types.

4.1 XYZ

Defines a point x units right, y units upward, and z units away.

x: number

Default: 0

The number of units in the x direction.

y: number

Default: 0

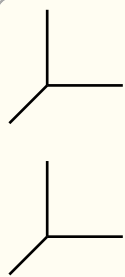
The number of units in the y direction.

z: number

Default: 0

The number of units in the z direction.

The implicit form can be given as an array of two or three number s, as in (x,y) and (x,y,z).



```
line((0,0), (x: 1))
line((0,0), (y: 1))
line((0,0), (z: 1))

// Implicit form
line((0, -2), (1, -2))
line((0, -2), (0, -1, 0))
line((0, -2), (0, -2, 1))
```

4.2 Previous

Use this to reference the position of the previous coordinate passed to a draw function. This will never reference the position of a coordinate used in to define another coordinate. It takes the form of an empty array (). The previous position initially will be (0, 0, 0).



4.3 Relative

Places the given coordinate relative to the previous coordinate. Or in other words, for the given coordinate, the previous coordinate will be used as the origin. Another coordinate can be given to act as the previous coordinate instead.

rel: coordinate

The coordinate to be place relative to the previous coordinate.

update: boolean

Default: true

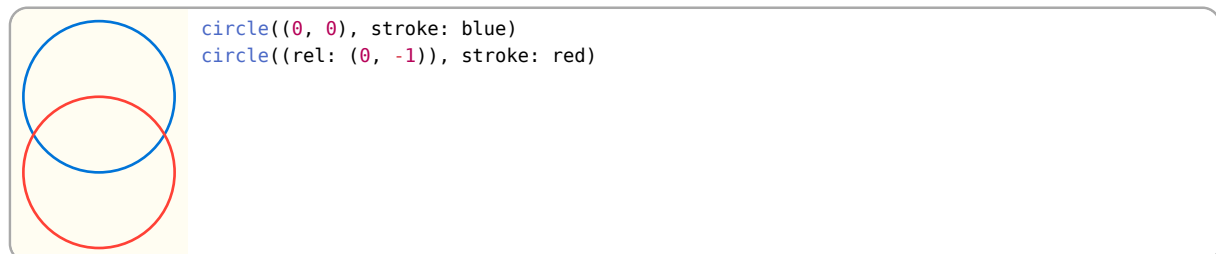
When false the previous position will not be updated.

to: coordinate

Default: ()

The coordinate to treat as the previous coordinate.

In the example below, the red circle is placed one unit below the blue circle. If the blue circle was to be moved to a different position, the red circle will move with the blue circle to stay one unit below.



4.4 Polar

Defines a point that is radius distance away from the origin at the given angle.

angle: angle

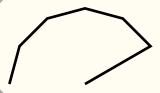
The angle of the coordinate. An angle of 0deg is to the right, a degree of 90deg is upward. See <https://typst.app/docs/reference/layout/angle/> for details.

radius: number or tuple<number>

The distance from the origin. An array can be given, in the form (x, y) to define the x and y radii of an ellipse instead of a circle.



The implicit form is an array of the angle then the radius (angle, radius) or (angle, (x, y)).



```
line((0,0), (30deg, 1), (60deg, 1),
      (90deg, 1), (120deg, 1), (150deg, 1), (180deg, 1))
```

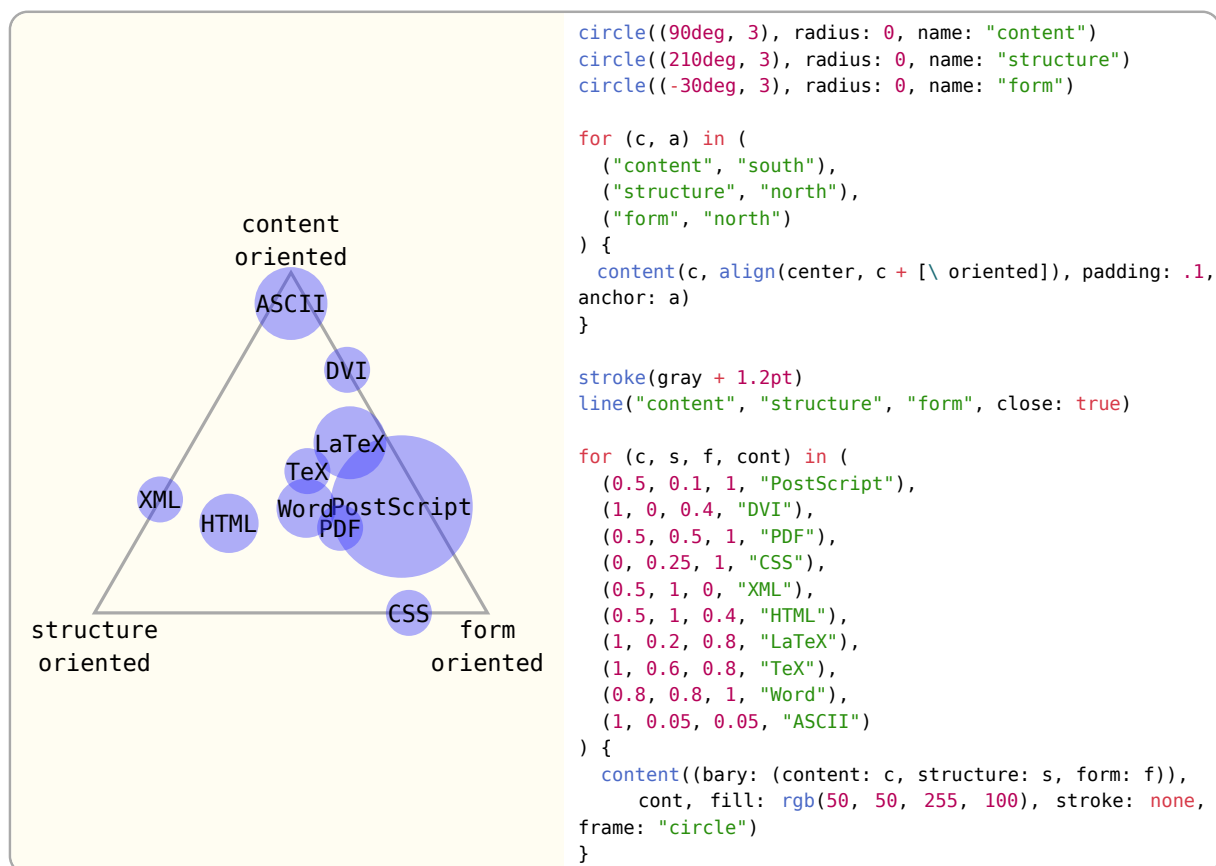
4.5 Barycentric

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2, \dots, v_n and numbers $\alpha_1, \alpha_2, \dots, \alpha_n$. Then the barycentric coordinate specified by these vectors and numbers is

$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \dots + \alpha_n}$$

bary: dictionary

A dictionary where the key is a named element and the value is a `float`. The center anchor of the named element is used as v and the value is used as a .



4.6 Anchor

Defines a point relative to a named element using anchors, see Section 2.2.

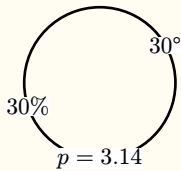
name: string

The name of the element that you wish to use to specify a coordinate.

anchor: number or angle or string or ratio or none

Default: none

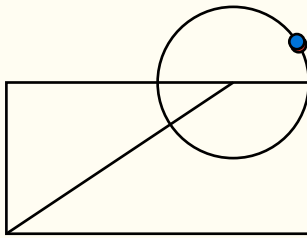
The anchor of the element. Strings are named anchors, angles are border anchors and numbers and ratios are path anchors. If not given, the default anchor will be used, on most elements this is center but it can be different.



```
circle((0,0), name: "circle")
// Anchor at 30 degree
content((name: "circle", anchor: 30deg), box(fill: white, $ 30 degree $))
// Anchor at 30% of the path length
content((name: "circle", anchor: 30%), box(fill: white, $ 30 % $))
// Anchor at 3.14 of the path
content((name: "circle", anchor: 3.14), box(fill: white, $ p = 3.14 $))
```

Note, that not all elements provide border or path anchors!

You can also use implicit syntax of a dot separated string in the form "name.anchor" for all anchors. Because named elements in groups act as anchors, you can also access them through this syntax.



```
group(name: "group", {
  line((0,0), (3,2), name: "line")
  circle("line.end", name: "circle")
  rect("line.start", "circle.east")

  circle("circle.30deg", radius: 0.1, fill: red)
})

circle("group.circle.-1", radius: 0.1, fill: blue)
```

4.7 Tangent

This system allows you to compute the point that lies tangent to a shape. In detail, consider an element and a point. Now draw a straight line from the point so that it “touches” the element (more formally, so that it is *tangent* to this element). The point where the line touches the shape is the point referred to by this coordinate system.

element: string

The name of the element on whose border the tangent should lie.

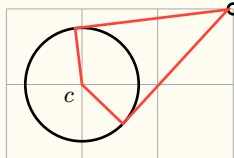
point: coordinate

The point through which the tangent should go.

solution: integer

Which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently it does this by assuming the distance between the center and top anchor (See Section 2.2) is the radius of a circle.



```
grid((0,0), (3,2), help-lines: true)

circle((3,2), name: "a", radius: 2pt)
circle((1,1), name: "c", radius: 0.75)
content("c", $ c $, anchor: "north-east", padding: .1)

stroke(red)
line("a", (element: "c", point: "a", solution: 1),
      "c", (element: "c", point: "a", solution: 2),
      close: true)
```

4.8 Perpendicular

Can be used to find the intersection of a vertical line going through a point p and a horizontal line going through some other point q .

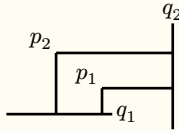
horizontal: coordinate

The coordinate through which the horizontal line passes.

vertical: coordinate

The coordinate through which the vertical line passes.

You can use the implicit syntax of (horizontal, "-|", vertical) or (vertical, "|-", horizontal)



```
set-style(content: (padding: .05))
content((30deg, 1), $ p_1 $, name: "p1")
content((75deg, 1), $ p_2 $, name: "p2")

line((-0.2, 0), (1.2, 0), name: "xline")
content("xline.end", $ q_1 $, anchor: "west")
line((2, -0.2), (2, 1.2), name: "yline")
content("yline.end", $ q_2 $, anchor: "south")

line("p1.south-east", (horizontal: ()), vertical: "xline.end"))
line("p2.south-east", ((, "|-", "xline.end")) // Short form
line("p1.south-east", (vertical: ()), horizontal: "yline.end"))
line("p2.south-east", ((, "-|", "yline.end")) // Short form
```

4.9 Interpolation

Use this to linearly interpolate between two coordinates **a** and **b** with a given distance number. If number is a **number** the position will be at the absolute distance away from **a** towards **b**, a **ratio** can be given instead to be the relative distance between **a** and **b**. An angle can also be given for the general meaning: “First consider the line from **a** to **b**. Then rotate this line by angle around point **a**. Then the two endpoints of this line will be **a** and some point **c**. Use this point **c** for the subsequent computation.”

a: coordinate

The coordinate to interpolate from.

b: coordinate

The coordinate to interpolate to.

number: **ratio** or **number**

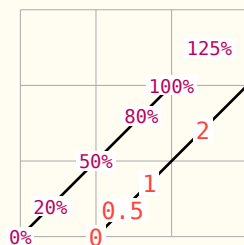
The distance between **a** and **b**. A ratio will be the relative distance between the two points, a number will be the absolute distance between the two points.

angle: **angle**

Default: 0deg

Angle between \overrightarrow{AB} and \overrightarrow{AP} , where **P** is the resulting coordinate. This can be used to get the *normal* for a tangent between two points.

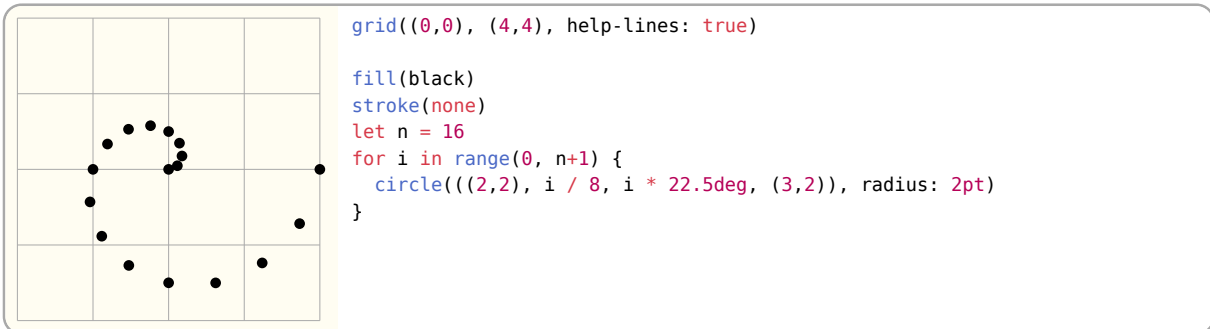
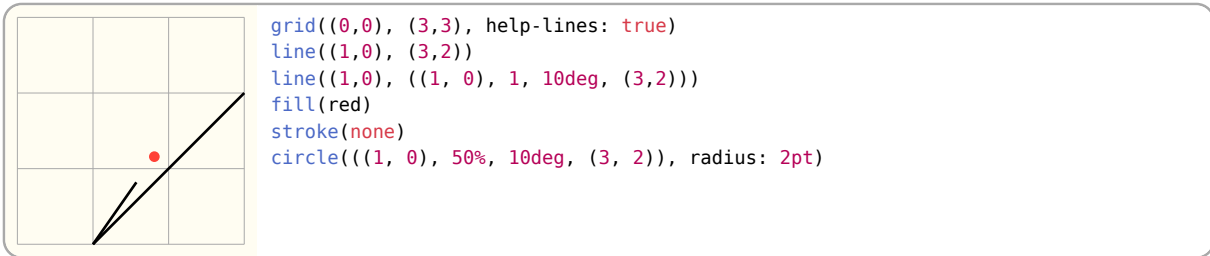
Can be used implicitly as an array in the form (**a**, **number**, **b**) or (**a**, **number**, **angle**, **b**).



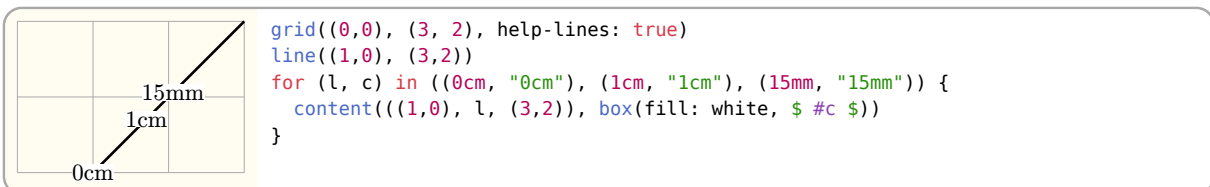
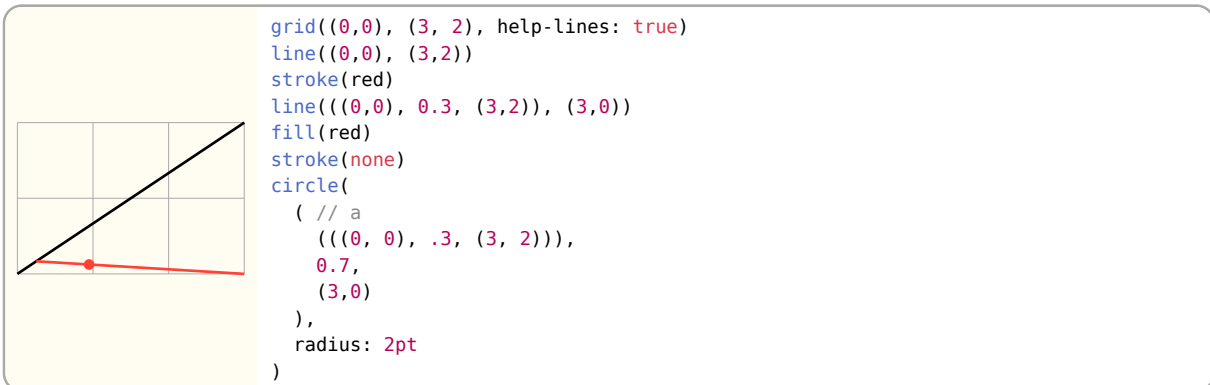
```
grid((0,0), (3,3), help-lines: true)

line((0,0), (2,2), name: "a")
for i in (0%, 20%, 50%, 80%, 100%, 125%) { /* Relative distance */
  content(("a.start", i, "a.end"),
    box(fill: white, inset: 1pt, [#i]))
}

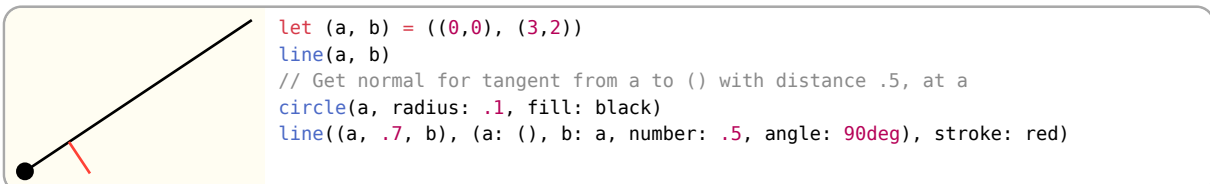
line((1,0), (3,2), name: "b")
for i in (0, 0.5, 1, 2) { /* Absolute distance */
  content(("b.start", i, "b.end"),
    box(fill: white, inset: 1pt, text(red, [#i])))
}
```

You can even chain them together!



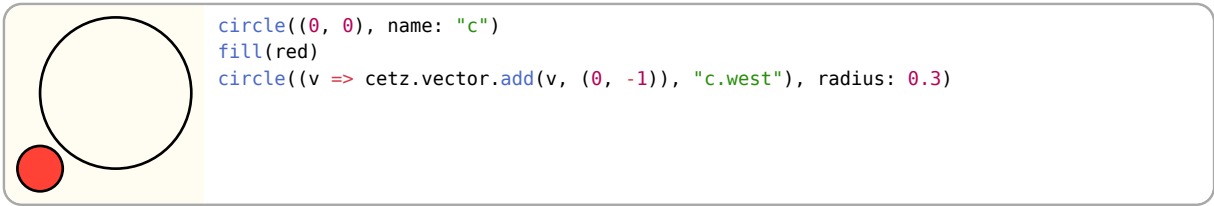
Interpolation coordinates can be used to get the *normal* on a tangent:



4.10 Function

An array where the first element is a function and the rest are coordinates will cause the function to be called with the resolved coordinates. The resolved coordinates have the same format as the implicit form of the 3-D XYZ coordinate system, Section 4.1.

The example below shows how to use this system to create an offset from an anchor, however this could easily be replaced with a relative coordinate with the `to` argument set, Section 4.3.



5 Libraries

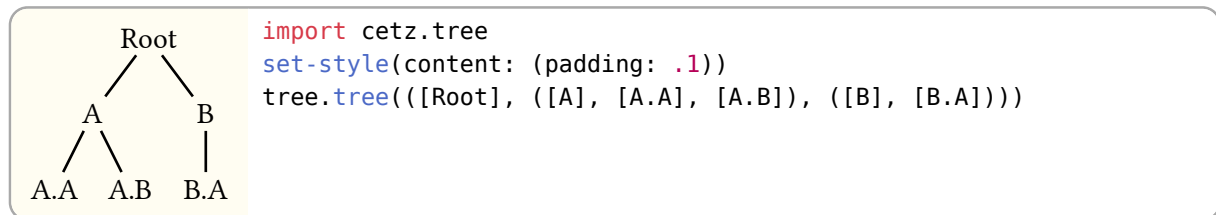
5.1 Tree

The tree library allows the drawing diagrams with simple tree layout algorithms

5.1.1 tree

Lays out and renders tree nodes.

For each node, the tree function creates an anchor of the format "node-<depth>-<child-index>" that can be used to query a nodes position on the canvas.



Parameters

```
tree(
  root: array,
  draw-node: auto function,
  draw-edge: none auto function,
  direction: string,
  parent-position: string,
  grow: float,
  spread: float,
  name: none string
)
```

root: array

A nested array of content that describes the structure the tree should take. Example: ([root], [child 1], ([child 2], [grandchild 1]))

draw-node: auto or function

Default: auto

The function to call to draw a node. The function will be passed two positional arguments, the node to draw and the node's parent, and is expected to return elements ((node, parent-node) => elements). The node's position is accessible through the "center" anchor or by using the previous position coordinate (). If auto is given, just the node's value will be drawn as content. The following predefined styles can be used:

draw-edge: none or auto or function

Default: auto

The function to call draw an edge between two nodes. The function will be passed the name of the starting node, the name of the ending node, and the end node and is expected to return elements ((source-name, target-name, target-node) => elements). If auto is given, a straight line will be drawn between nodes.

direction: string

Default: "down"

A string describing the direction the tree should grow in ("up", "down", "left", "right")

parent-position: string

Default: "center"

Positioning of parent nodes (begin, center, end)

grow: float

Default: 1

Depth grow factor

spread: float

Default: 1

Sibling spread factor

name: none or string

Default: none

The tree elements name

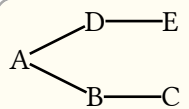
Nodes

A tree node is an array consisting of the nodes value at index 0 followed by its child nodes. For the default draw-node function, the value (first item) of a node must be of type `content`.

Example of a list of nodes:

A—B—C—D `cetz.tree.tree([A], ([B], ([C], ([D],)))), direction: "right")`

Example of a tree of nodes:

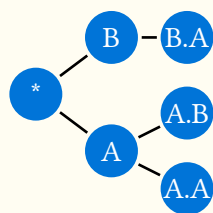
 `cetz.tree.tree([A], ([B], [C]), ([D], [E])), direction: "right")`

Drawing and Styling Tree Nodes

The `tree()` function takes an optional `draw-node:` and `draw-edge:` callback function that can be used to customize node and edge drawing.

The `draw-node` function must take the current node and its parents node anchor as arguments and return one or more elements.

For drawing edges between nodes, the `draw-edge` function must take two node anchors and the target node as arguments and return one or more elements.



```
import cetz.tree
let data = ([\*, ([A], [A.A], [A.B]), ([B], [B.A]))
tree.tree(
  data,
  direction: "right",
  draw-node: (node, ..) => {
    circle((), radius: .35, fill: blue, stroke: none)
    content((), text(white, [#node.content]))
  },
  draw-edge: (from, to, ..) => {
    let (a, b) = (from + ".center", to + ".center")
    line((a, .4, b), (b, .4, a))
  }
)
```

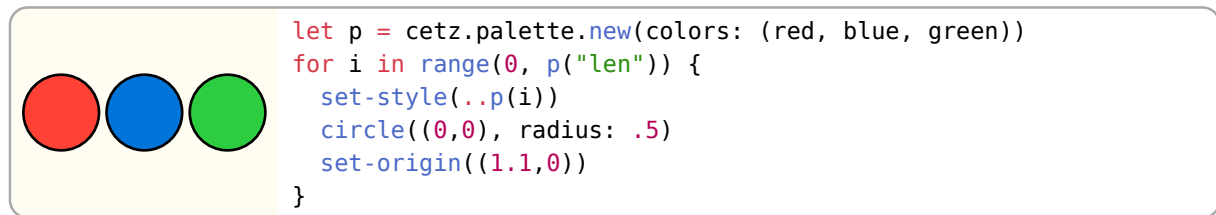
5.2 Palette

A palette is a function of the form `index => style` that takes an index, that can be any integer and returns a canvas style dictionary. If passed the string "len" it must return the length of its unique styles. An example use for palette functions is the plot library, which can use palettes to apply different styles per plot.

The palette library provides some predefined palettes.

5.2.1 new

Create a new palette based on a base style



The functions returned by this function have the following named arguments:

fill: bool Default: **true**

If true, the returned fill color is one of the colors from the colors list, otherwise the base styles fill is used.

stroke: bool Default: **false**

If true, the returned stroke color is one of the colors from the colors list, otherwise the base styles stroke color is used.

You can use a palette for stroking via: `red.with(stroke: true)`.

Parameters

```
new(
  base: style,
  colors: none array,
  dash: none array
)-> function Palette function of the form `index => style` that returns a style for an integer index
```

base: style Default: base-style

Style dictionary to use as base style for the styles generated per color

colors: none or array Default: ()

List of colors the returned palette should return styles with

dash: none or array Default: ()

List of stroke dash patterns the returned palette should return styles with

5.2.2 List of predefined palettes

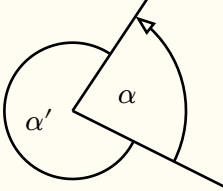
- gray
- red
- orange
- light-green
- dark-green
- turquoise
- cyan
- blue
- indigo
- purple
- magenta
- pink
- rainbow
- tango-light
- tango
- tango-dark

5.3 Angle

The angle function of the angle module allows drawing angles with an optional label.

5.3.1 angle

Draw an angle between a and b through origin origin



```

line((0,0), (1,1.5), name: "a")
line((0,0), (2,-1), name: "b")

// Draw an angle between the two lines
cetz.angle.angle("a.start", "a.end", "b.end", label: $ alpha $,
  mark: (end: ">"), radius: 1.5)
cetz.angle.angle("a.start", "b.end", "a.end", label: $ alpha' $,
  radius: 50%, inner: false)

```

Style Root: angle

Style Keys:

radius: number

Default: 0.5

The radius of the angles arc. If of type ratio, it is relative to the smaller distance of either origin to a or origin to b.

label-radius: number or ratio

Default: 50%

The radius of the angles label origin. If of type ratio, it is relative to radius.

Anchors

"a" Point a

"b" Point b

"origin" Origin

"label" Label center

"start" Arc start

"end" Arc end

Parameters

```

angle(
  origin: coordinate,
  a: coordinate,
  b: coordinate,
  inner: bool,
  label: none content function,
  name: none string,
  ..style: style
)

```

origin: coordinate

Angle origin

a: coordinate

Coordinate of side a, containing an angle between origin and b.

b: coordinate

Coordinate of side b, containing an angle between origin and a.

inner: bool

Default: true

Draw the smaller (inner) angle if true, otherwise the outer angle gets drawn.

label: `none` or `content` or `function`

Default: `none`

Draw a label at the angles "label" anchor. If label is a function, it gets the angle value passed as argument. The function must be of the format `angle => content`.

name: `none` or `string`

Default: `none`

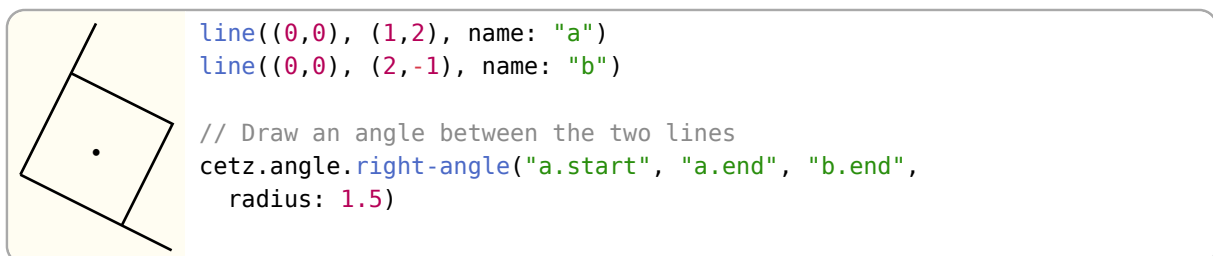
Element name, used for querying anchors.

..style: `style`

Style key-value pairs.

5.3.2 right-angle

Draw a right angle between a and b through origin origin



Style Root: `angle`

Style Keys:

radius: `number`

Default: `0.5`

The radius of the angles arc. If of type `ratio`, it is relative to the smaller distance of either origin to a or origin to b.

label-radius: `number` or `ratio`

Default: `50%`

The radius of the angles label origin. If of type `ratio`, it is relative to the distance between origin and the angle corner.

Aliases

"a" Point a

"b" Point b

"origin" Origin

"corner" Angle corner

"label" Label center

Parameters

```

right-angle(
  origin: coordinate,
  a: coordinate,
  b: coordinate,
  label: none content,
  name: none string,
  ..style: style
)

```

origin: `coordinate`

Angle origin

a: `coordinate`

Coordinate of side a, containing an angle between origin and b.

b: `coordinate`

Coordinate of side b, containing an angle between origin and a.

label: `none` or `content`

Default: `"•"`

Draw a label at the angles "label" anchor.

name: `none` or `string`

Default: `none`

Element name, used for querying anchors.

..style: `style`

Style key-value pairs.

Default angle Style

```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 50%,
  mark: auto,
)
```

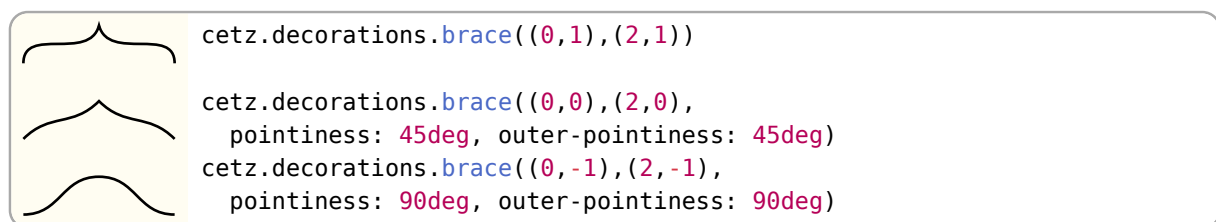
5.4 Decorations

Various pre-made shapes and path modifications.

5.4.1 Braces

5.4.2 brace

Draw a curly brace between two points.



Style Root: brace.

Style Keys:

amplitude: `number`

Default: `0.5`

Sets the height of the brace, from its baseline to its middle tip.

pointiness: `ratio` or `angle`

Default: `15deg`

How pointy the spike should be. `0deg` or 100% for maximum pointiness, `90deg` or 0% for minimum.

outer-pointiness: `ratio` or `angle`

Default: `15deg`

How pointy the outer edges should be. `0deg` or 100% for maximum pointiness (allowing for a smooth transition to a straight line), `90deg` or 0% for minimum. Setting this to `auto` will use the value set for pointiness.

content-offset: `number`Default: `0.3`

Offset of the "content" anchor from the spike of the brace.

flip: `bool`Default: `false`

Mirror the brace along the line between start and end

Anchors:**start** Where the brace starts, same as the start parameter.**end** Where the brace end, same as the end parameter.**spike** Point of the spike, halfway between start and end and shifted by amplitude towards the pointing direction.**content** Point to place content/text at, in front of the spike.**center** Center of the enclosing rectangle.**Parameters**

```
brace(
  start: coordinate,
  end: coordinate,
  ..style: style,
  name: string none
)
```

start: `coordinate`

Start point

end: `coordinate`

End point

..style: `style`

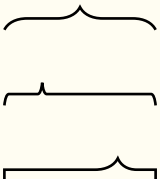
Style key-value pairs

name: `string` or `none`Default: `none`

Element name used for querying anchors

5.4.3 flat-brace

Draw a flat curly brace between two points.



```
cetz.decorations.flat-brace((0,1),(2,1))
cetz.decorations.flat-brace((0,0),(2,0),
  curves: .2,
  aspect: 25%)
cetz.decorations.flat-brace((0,-1),(2,-1),
  outer-curves: 0,
  aspect: 75%)
```

This mimics the braces from TikZ's `decorations.pathreplacing` library¹. In contrast to `brace()`, these braces use straight line segments, resulting in better looks for long braces with a small amplitude.

¹<https://github.com/pgf-tikz/pgf/blob/6e5fd71581ab04351a89553a259b57988bc28140/tex/generic/pgf/libraries/decorations/pgflibrarydecorations.pathreplacing.code.tex#L136-L185>

Style Root: flat-brace

Style Keys:

amplitude: number Default: 0.3

Determines how much the brace rises above the base line.

aspect: ratio Default: 50%

Determines the fraction of the total length where the spike will be placed.

curves: number Default: auto

Curviness factor of the brace, a factor of 0 means no curves.

outer-curves: auto or number Default: auto

Curviness factor of the outer curves of the brace. A factor of 0 means no curves.

Anchors:

start Where the brace starts, same as the start parameter.

end Where the brace end, same as the end parameter.

spike Point of the spike's top.

content Point to place content/text at, in front of the spike.

center Center of the enclosing rectangle.

Parameters

```
flat-brace(
  start: coordinate,
  end: coordinate,
  flip: bool,
  debug,
  name: string none,
  ..style: style
)
```

start: coordinate

Start point

end: coordinate

End point

flip: bool Default: false

Flip the brace around

debug: Default: false

name: string or none Default: none

Element name for querying anchors

..style: style

Style key-value pairs

5.4.4 Path Decorations

Path decorations are elements that accept a path as input and generate one or more shapes that follow that path.

All path decoration functions support the following style keys:

start	<code>ratio</code> or <code>length</code>	(default: <code>0%</code>)
	Absolute or relative start of the decoration on the path.	
stop	<code>ratio</code> or <code>length</code>	(default: <code>100%</code>)
	Absolute or relative end of the decoration on the path.	
rest	<code>string</code>	(default: <code>LINE</code>)
	If set to "LINE", generate lines between the paths start/end and the decorations start/end if the path is <i>not closed</i> .	
width	<code>number</code>	(default: <code>1</code>)
	Width or thickness of the decoration.	
segments	<code>int</code>	(default: <code>10</code>)
	Number of repetitions/phases to generate. This key is ignored if <code>segment-length</code> is set <code>!= none</code> .	
segment-length	<code>none</code> or <code>number</code>	
	Length of one repetition/phase of the decoration.	
align	<code>"START"</code> , <code>"MID"</code> , <code>"END"</code>	(default: <code>START</code>)
	Alignment of the decoration on the path <i>if segment-length is set</i> and the decoration does not fill up the full range between start and stop.	

5.4.5 zigzag

Draw a zig-zag or saw-tooth wave along a path

The number of teeth can be controlled via the `segments` or `segment-length` style key, and the width via amplitude.



```
line((0,0), (2,1), stroke: gray)
cetz.decorations.zigzag(line((0,0), (2,1)), amplitude: .25, start: 10%, stop: 90%)
```

Styling

Root zigzag

Keys

factor:	<code>ratio</code>	Default: <code>100%</code>
	Triangle mid between its start and end. Setting this to <code>0%</code> leads to a falling sawtooth shape, while <code>100%</code> results in a raising sawtooth	

Parameters

```
zigzag(
  target: drawable,
  name: none string,
  close: auto bool,
  ..style: style
)
```

target: `drawable`

Target path

name: `none` or `string` Default: `none`

Element name

close: `auto` or `bool` Default: `auto`

Close the path

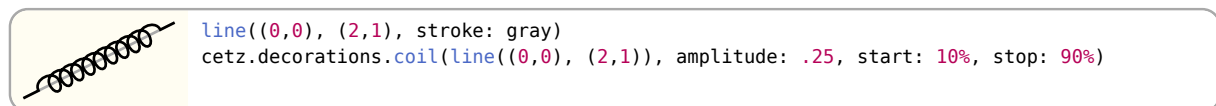
..style: style

Style

5.4.6 coil

Draw a stretched coil/loop spring along a path

The number of windings can be controlled via the `segments` or `segment-length` style key, and the width via `amplitude`.



Styling

Root coil

Keys

factor: ratio

Default: 150%

Factor of how much the coil overextends its length to form a curl.

Parameters

```
coil(
  target: drawable,
  close: auto bool,
  name: none string,
  ..style: style
)
```

target: drawable

Target path

close: auto or bool

Default: auto

Close the path

name: none or string

Default: none

Element name

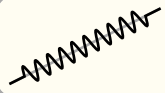
..style: style

Style

5.4.7 wave

Draw a wave along a path using a catmull-rom curve

The number of phases can be controlled via the `segments` or `segment-length` style key, and the width via `amplitude`.



```
line((0,0), (2,1), stroke: gray)
cetz.decorations.wave(line((0,0), (2,1)), amplitude: .25, start: 10%, stop: 90%)
```

Styling

Root wave

Keys

tension: float

Default: 0.5

Catmull-Rom curve tension, see `catmull()`.

Parameters

```
wave(
  target: drawable,
  close: auto bool,
  name: none string,
  ..style: style
)
```

target: drawable

Target path

close: auto or bool

Default: auto

Close the path

name: none or string

Default: none

Element name

..style: style

Style

Styling

Default brace Style

```
(
  amplitude: 0.5,
  pointiness: 15deg,
  outer-pointiness: 0deg,
  content-offset: 0.3,
  flip: false,
  stroke: auto,
  fill: none,
)
```

Default flat-brace Style

```
(
  amplitude: 0.3,
  aspect: 50%,
  curves: (1, 0.5, 0.6, 0.15),
  outer-curves: auto,
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

6 Advanced Functions

6.1 Coordinate

6.1.1 resolve

Resolve a list of coordinates to absolute vectors

(1.0, 1.0, 0.0)
(0.0, 0.0, 0.0)

```
line((0,0), (1,1), name: "l")
get-ctx(ctx => {
  // Get the vector of coordinate "l.start" and "l.end"
  let (ctx, a, b) = cetz.coordinate.resolve(ctx, "l.start",
    "l.end")
  content("l.start", [#a], frame: "rect", stroke: none, fill:
    white)
  content("l.end",    [#b], frame: "rect", stroke: none, fill:
    white)
})
```

Parameters

```
resolve(
  ctx: context,
  ..coordinates: coordinate,
  update: bool
)-> (ctx vector..) Returns a list of the new context object plus the
```

ctx: context

Canvas context object

..coordinates: coordinate

List of coordinates

update: bool

Default: **true**

Update the context's last position resolved coordinate vectors

6.2 Styles

6.2.1 resolve

You can use this to combine the style in ctx, the style given by a user for a single element and an element's default style.

base is first merged onto dict without overwriting existing values, and if root is given it is merged onto that key of dict. merge is then merged onto dict but does overwrite existing entries, if root is given it is merged onto that key of dict. Then entries in dict that are **auto** inherit values from their nearest ancestor and entries of type **dictionary** are merged with their closest ancestor.

```
#let dict = (
  stroke: "black",
  fill: none,
  mark: (stroke: auto, fill: "blue"),
  line: (stroke: auto, mark: auto, fill: "red")
)
#styles.resolve(dict, merge: (mark: (stroke: "yellow")), root: "line")

(
  stroke: "black",
  mark: (stroke: "yellow", fill: "blue"),
  fill: "red",
)
```

The following is a more detailed explanation of how the algorithm works to use as a reference if needed. It should be updated whenever changes are made. Remember that dictionaries are recursively merged, if an entry is any other type it is simply updated. (dict + dict = merged dict, value + dict = dict, dict + value = value) First if base is given, it will be merged without overwriting values onto dict. If root is given it will be merged onto that key of dict. Each level of dict is then processed with these steps. If root is given the level with that key will be the first, otherwise the whole of dict is processed.

1. Values on the corresponding level of merge are inserted into the level if the key does not exist on the level or if they are not both dictionaries. If they are both dictionaries their values will be inserted in the same stage at a lower level.
2. If an entry is auto or a dictionary, the tree is travelled back up until an entry with the same key is found. If the current entry is auto the value of the ancestor's entry is copied. Or if the current entry and ancestor entry is a dictionary, they are merged with the current entry overwriting any values in it's ancestors.
3. Each entry that is a dictionary is then resolved from step 1.

```
(
  scale: 1,
  length: 5.67pt,
  width: 4.25pt,
  inset: 1.42pt,
  sep: 2.83pt,
  pos: none,
  offset: 0,
  start: none,
  end: none,
  symbol: none,
  xy-up: (0, 0, 1),
  z-up: (0, 1, 0),
  stroke: 1pt + luma(0%),
  fill: none,
  slant: none,
  harpoon: false,
  flip: false,
  reverse: false,
  flex: true,
  position-samples: 30,
  shorten-to: auto,
  transform-shape: true,
  anchor: "tip",
)

get-ctx(ctx => {
  // Get the current "mark" style
  content((0,0), [#cetz.styles.resolve(ctx.style, root:
    "mark")])
})
```

Parameters

```
resolve(
  dict: style,
  root: none str,
  merge: style,
  base: none style
)
```

dict: style

Current context style (ctx.style).

root: none or str

Style root element name.

Default: none

merge: `style`Default: `(:)`

Style values overwriting the current style. I.e. inline styles passed with an element: `line(..., stroke: red)`.

base: `none` or `style`Default: `(:)`

Style values to merge into dict without overwriting it.

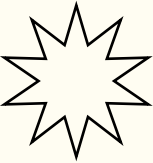
6.2.2 Default Style

This is a dump of the style dictionary every canvas gets initialized with. It contains all supported keys for all elements.

```
(
  fill: none,
  stroke: 1pt + luma(0%),
  radius: 1,
  shorten: "LINEAR",
  padding: none,
  mark: (
    scale: 1,
    length: 5.67pt,
    width: 4.25pt,
    inset: 1.42pt,
    sep: 2.83pt,
    pos: none,
    offset: 0,
    start: none,
    end: none,
    symbol: none,
    xy-up: (0, 0, 1),
    z-up: (0, 1, 0),
    stroke: auto,
    fill: auto,
    slant: none,
    harpoon: false,
    flip: false,
    reverse: false,
    flex: true,
    position-samples: 30,
    shorten-to: auto,
    transform-shape: true,
    anchor: "tip",
  ),
  circle: (radius: auto, stroke: auto, fill: auto),
  group: (padding: auto, fill: auto, stroke: auto),
  line: (mark: auto, fill: auto, stroke: auto),
  bezier: (
    stroke: auto,
    fill: auto,
    mark: auto,
    shorten: auto,
  ),
  catmull: (
    tension: 0.5,
    mark: auto,
    shorten: auto,
    stroke: auto,
    fill: auto,
  ),
  hobby: (
    omega: (1, 1),
    mark: auto,
    shorten: auto,
    stroke: auto,
    fill: auto,
  ),
  rect: (radius: 0, stroke: auto, fill: auto),
  arc: (
    mode: "OPEN",
    update-position: true,
    mark: auto,
    stroke: auto,
    fill: auto,
    radius: auto,
  ),
  ),
  content: (
    padding: auto,
    frame: none,
    fill: auto,
    stroke: auto,
  ),
  prism: (stroke: auto, fill: auto),
)
```


7 Creating Custom Elements

The simplest way to create custom, reusable elements is to return them as a group. In this example we will implement a function `my-star(center)` that draws a star with `n` corners and a style specified inner and outer radius.



```
let my-star(center, name: none, ..style) = {
  group(name: name, ctx => {
    // Define a default style
    let def-style = (n: 5, inner-radius: .5, radius: 1)

    // Resolve the current style ("star")
    let style = cetz.styles.resolve(ctx.style, merge: style.named(),
      base: def-style, root: "star")

    // Compute the corner coordinates
    let corners = range(0, style.n * 2).map(i => {
      let a = 90deg + i * 360deg / (style.n * 2)
      let r = if calc.rem(i, 2) == 0 { style.radius } else { style.inner-radius }

      // Output a center relative coordinate
      (rel: (calc.cos(a) * r, calc.sin(a) * r, 0), to: center)
    })

    line(..corners, ..style, close: true)
  })
}

// Call the element
my-star((0,0))
my-star((0,3), n: 10)

set-style(star: (fill: yellow)) // set-style works, too!
my-star((0,6), inner-radius: .3)
```

8 Internals

8.1 Context

The state of the canvas is encoded in its context dictionary. Elements or other draw calls may return a modified context to the canvas to change its state, e.g. modifying the transforming matrix, adding a group or setting a style.

The context can be manually retrieved and modified using the `get-ctx` and `set-ctx` functions.

8.2 Elements

Each CeTZ element (`line`, `bezier`, `circle`, ...) returns an array of functions for drawing to the canvas. Such function takes the canvas' context and must return an dictionary of the following keys:

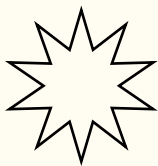
- `ctx` (required): The (modified) canvas context object
- `drawables`: List of drawables to render to the canvas
- `anchors`: A function of the form (`<anchor-identifier>`) => `<vector>`
- `name`: The elements name

An element that does only modify the context could be implemented like the following:

```
let my-element() = {
  (ctx => {
    // Do something with ctx ...
    (ctx: ctx)
  },)
}

// Call the element
my-element()
```

For drawing, elements must not use Typst native drawing functions, but output CeTZ paths. The `drawable` module provides functions for path creation (`path(...)`), the `path-util` module provides utilities for path segment creation. For demonstration, we will recreate the custom element `my-star` from Section 7:



```
import cetz.drawables: path
import cetz.vectors

let my-star(center, ..style) = {
  (ctx => {
    // Define a default style
    let def-style = (n: 5, inner-radius: .5, radius: 1, stroke: auto, fill: auto)

    // Resolve center to a vector
    let (ctx, center) = cetz.coordinate.resolve(ctx, center)

    // Resolve the current style ("star")
    let style = cetz.styles.resolve(ctx.style, merge: style.named(),
      base: def-style, root: "star")

    // Compute the corner coordinates
    let corners = range(0, style.n * 2).map(i => {
      let a = 90deg + i * 360deg / (style.n * 2)
      let r = if calc.rem(i, 2) == 0 { style.radius } else { style.inner-radius }
      vector.add(center, (calc.cos(a) * r, calc.sin(a) * r, 0))
    })

    // Build a path through all three coordinates
    let path = cetz.drawables.path((cetz.path-util.line-segment(corners)),
      stroke: style.stroke, fill: style.fill, close: true)

    (ctx: ctx,
     drawables: cetz.drawables.apply-transform(ctx.transform, path),
    ),)
  },)
}

// Call the element
my-star((0,0))
my-star((0,3), n: 10)
my-star((0,6), inner-radius: .3, fill: yellow)
```

Using custom elements instead of groups (as in Section 7) makes sense when doing advanced computations or even applying modifications to passed in elements.