The CeTZ package

Johannes Wolf and fenjalien
https://github.com/johannes-wolf/typst-canvas

Contents

1. Introduction

This package provides a way to draw stuff 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 Zeichenpacket" (german for "CeTZ, a Typst drawing package") and is pronounced like the word "Cats".

2. Usage

This is the minimal starting point:

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  ...
})
```

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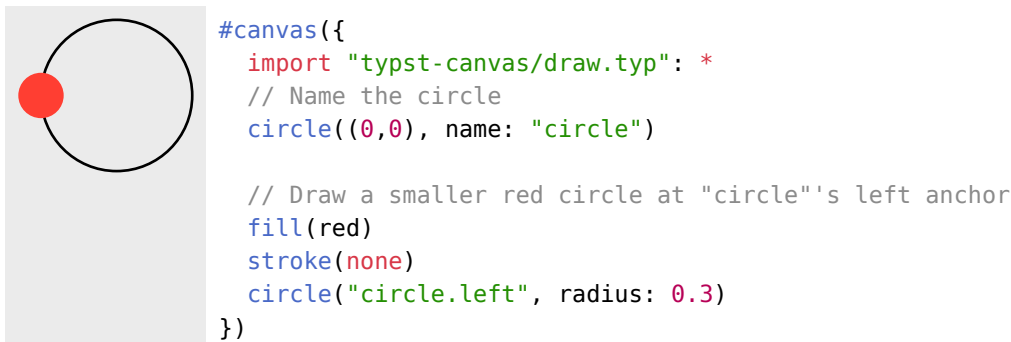
2.1. Argument Types

Argument types in this document are formatted in `monospace` and encased in angle brackets <>. Types such as `<integer>` and `<content>` are the same as Typst but additional are required:

`<coordinate>` Any coordinate system. See Section 4.

**`<number>`** `<integer>` or `<float>`

2.2. Anchors

Anchors are named positions relative to named elements.

To use an anchor of an element, you must give the element a name using the `name` argument.

```
#canvas({
  import "typst-canvas/draw.typ": *
  // Name the circle
  circle((0,0), name: "circle")

  // Draw a smaller red circle at "circle"'s left anchor
  fill(red)
  stroke(none)
  circle("circle.left", radius: 0.3)
})
```

All elements will have default anchors based on their bounding box, they are: `center`, `left`, `right`, `above`/`top` and `below`/`bottom`, `top-left`, `top-right`, `bottom-left`, `bottom-right`. Some elements will have their own anchors.

Elements can be placed relative to their own anchors.
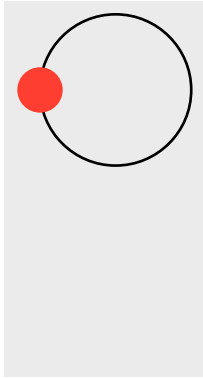
```
#canvas({
  import "typst-canvas/draw.typ": *
  // An element does not have to be named
  // in order to use its own anchors.
  circle((0,0), anchor: "left")

  // Draw a smaller red circle at the origin
  fill(red)
  stroke(none)
  circle((0,0), radius: 0.3)
})
```

3. Draw Function Reference

3.1. Canvas

`#canvas(background: none, length: 1cm, debug: false, body)`

background `<color>`   (default: none)
>   A color to be used for the background of the canvas.

length `<length>`   (default: 1cm)
>   Used to specify what 1 coordinate unit is.

debug `<bool>`   (default: false)
>   Shows the bounding boxes of each element when `true`.

body
>   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 have stroke and fill styling unless said otherwise.

fill `<color>` or `<none>`   (default: none)
>   How to fill the draw element.

stroke `<none>` or `<auto>` or `<length>`   (default: black + 1pt)
>   or `<color>` or `<dicitionary>` or `<stroke>`
>   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

```
#cetz.canvas({
  import cetz.draw: *
  // Draws a red circle with a blue border
  circle((0, 0), fill: red, stroke: blue)
  // Draws a green line
  line((0, 0), (1, 1), stroke: green)
})
```

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.
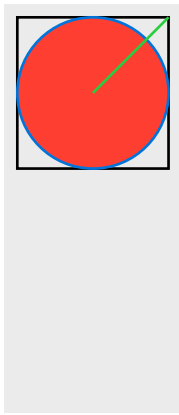
```
#cetz.canvas({
  import cetz.draw: *
  // Draws an empty square with a black border
  rect((-1, -1), (1, 1))

  // 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((), (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).
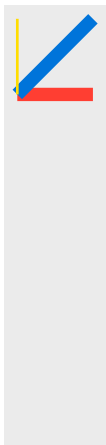
```
#canvas({
  import cetz.draw: *
  // Sets the stroke to red with a thickness of 5pt
  set-style(stroke: (paint: red, thickness: 5pt))
  // Draws a line with the global stroke
  line((0,0), (1,0))
  // Draws a blue line with a thickness of 5pt because dictionaries update the
style
  line((0,0), (1,1), 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.
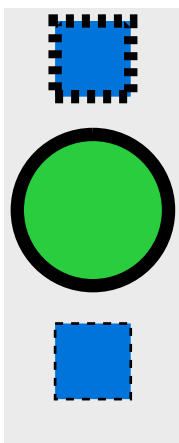
```
#canvas({
  import cetz.draw: *
  set-style(
    // Global fill and stroke
    fill: green,
    stroke: (thickness: 5pt),
    // Stroke and fill for only rectangles
    rect: (stroke: (dash: "dashed"), fill: blue),
  )
  rect((0,0), (1,1))
  circle((0.5, -1.5))
  rect((0,-3), (1, -4), stroke: (thickness: 1pt))
})
```

```
                  // Its a nice drawing okay
                  #cetz.canvas({
                    import cetz.draw: *
                    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)
                  })
```

3.3. Elements

3.3.1. Line

Draws a line (a direct path between two points) to the canvas. If multiplie coordinates are given, a line is drawn between each consecutive one.

```
#line(..pts, name: none, close: false, ..styling)
```

..pts `<arguments of coordinates>`
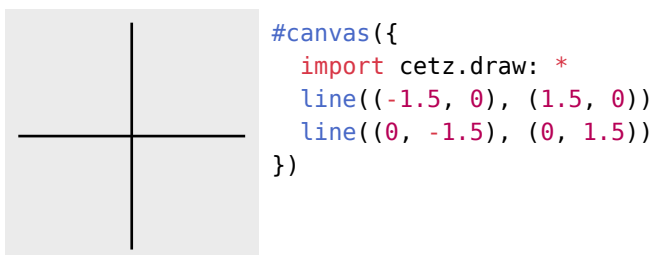   Coordinates to draw the lines between. A minimum of two must be given.

name `<string>`
    Sets the name of element for use with anchors.

close `<bool>`   (default: `false`)
    When `true` a straight line is drawn from the last coordinate to the first coordinate, essentially "closing" the shape.

```
#canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
})
```

Styling

mark `<dictionary>` or `<auto>`   (default: `auto`)
    The styling to apply to marks on the line, see Section 3.3.8.
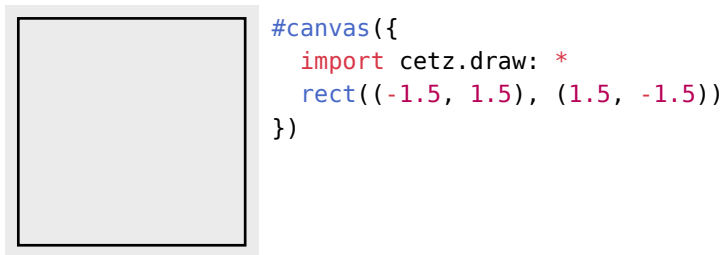
3.3.2. Rectangle

Draws a rectangle to the canvas.

```
#rect(a, b, name: none, anchor: none, ..styling)
```

a `<coordinate>`
  The top left coordinate of the rectangle.

b `<coordinate>`
  The bottom right coordinate of the rectangle.

```
#canvas({
  import cetz.draw: *
  rect((-1.5, 1.5), (1.5, -1.5))
})
```

3.3.3. Arc

Draws an arc to the canvas. Exactly two of the three values `start`, `stop`, and `delta` should be defined. You can set the radius of the arc by setting the `radius` style option. You can also draw an elliptical arc by passing an array where the first number is the radius in the x direction and the second number is the radius in the y direction.

```
#arc(position, start: auto, stop: auto, delta: auto, name: none, anchor: none,)
```

  position `<coordinate>`
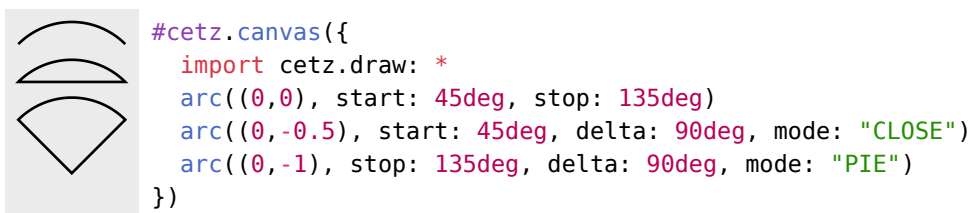   The coordinate to start drawing the arc from.

  start `<angle>`
    The angle to start the arc.

  stop `<angle>`
    The angle to stop the arc.

  delta `<angle>`
    The angle that is added to start or removed from stop.

```
#cetz.canvas({
  import cetz.draw: *
  arc((0,0), start: 45deg, stop: 135deg)
  arc((0,-0.5), start: 45deg, delta: 90deg, mode: "CLOSE")
  arc((0,-1), stop: 135deg, delta: 90deg, mode: "PIE")
})
```

Styling

  radius `<number>` or `<array>`   (default: 1)
    The radius of the arc. This is also a global style shared with circle!

  mode `<string>`   (default: "OPEN")
    The options are "OPEN" (the default, just the arc), "CLOSE" (a circular segment) and "PIE" (a circular sector).
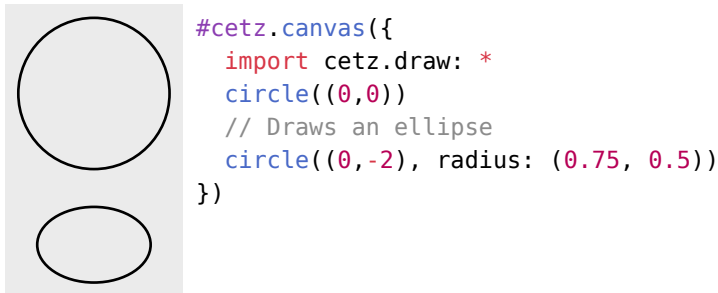
3.3.4. Circle

Draws a circle to the canvas. An ellipse can be drawn by passing an array of length two to the `radius` argument to specify its x and y radii.

```
#circle(center, name: none, anchor: none)
```

center `<coordinate>`
> The coordinate of the circle's origin.

```
#cetz.canvas({
  import cetz.draw: *
  circle((0,0))
  // Draws an ellipse
  circle((0,-2), radius: (0.75, 0.5))
})
```

Styling

radius `<number>` or `<length>` or `<array of <number> or <length>>` (default: 1)
> The circle's radius. If an array is given an ellipse will be drawn where the first item is the x radius and the second item is the y radius. This is also a global style shared with arc!

3.3.5. Bezier

Draws a bezier curve with 1 or 2 control points to the canvas.

`#bezier(start, end, ..ctrl-style)`

start `<coordinate>`
> The coordinate to start drawing the bezier curve from.

end `<coordinate>`
> The coordinate to draw the bezier curve to.

..ctrl-style `<coordinates>`
> An argument sink for the control points and styles. Its positional part should be of one or two coordinates to specify the control points of the bezier curve.

```
#cetz.canvas({
  import cetz.draw: *
  bezier((0, 0), (2, 0), (1, 1))
  bezier((0, -1), (2, -1), (.5, -2), (1.5, 0))
})
```
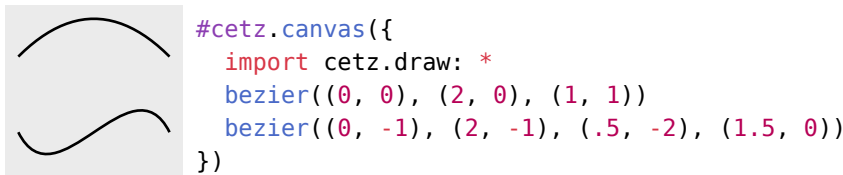
3.3.6. Content

Draws a content block to the canvas.

`#content(pt, ct, angle: 0deg, name: none, anchor: none)`

pt `<coordinate>`
> The coordinate of the center of the content block.

ct `<content>`
> The content block.

angle `<angle>`
> The angle to rotate the content block by. Uses Typst's `rotate` function.

```
Hello World!   #cetz.canvas({
                 import cetz.draw: *
                 content((0,0), [Hello World!])
               })
```

Styling

This draw element is not affected by fill or stroke styling.

padding `<length>`   (default: `0pt`)

3.3.7. Grid

Draws a grid to the canavas.

`#grid(from, to, step: 1, help-lines: false, name: none)`

from `<coordinate>`

   Specifies the bottom left position of the grid.

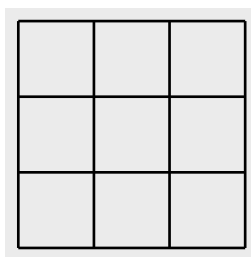**to** `<coordinate>`

   Specifies the top right position of the grid.

step `<number> or <length> or <array of <number> or <length>>`

   The stepping in both x and y directions. An array can be given to specify the stepping for each
   direction.

help-lines `<bool>`   (default: false)

   Styles the grid to look "subdued" by using thin gray lines (`0.2pt + gray`)

```
#cetz.canvas({
  import cetz.draw: *
  grid((0,0), (3,2), help-lines: true)
})
```

3.3.8. Mark

Draws a mark or "arrow head", its styling influences marks being drawn on paths (e.g. lines).

`#mark(from, to, ..style)`
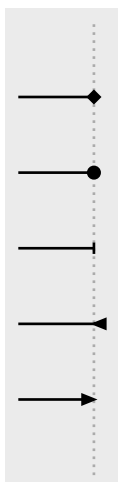
```
#cetz.canvas({
  import cetz.draw: *
  line((1, -1), (1, 5), stroke: (paint: gray, dash: "dotted"))
  set-style(mark: (fill: black))
  line((0, 4), (1, 4), mark: (end: "<>"))
  line((0, 3), (1, 3), mark: (end: "o"))
  line((0, 2), (1, 2), mark: (end: "|"))
  line((0, 1), (1, 1), mark: (end: "<"))
  line((0, 0), (1, 0), mark: (end: ">"))
})
```

Styling

symbol `<string>`   (default: >)

   The type of mark to draw when using the `mark` function.

start `<string>`

   The type of mark to draw at the start of a path.

end `<string>`

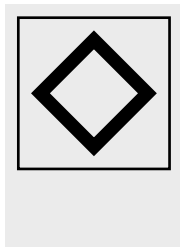> The type of mark to draw at the end of a path.

size `<number>` (default: 0.15)

> The size of the marks.

3.4. Groups

Groups allow scoping context changes such as setting stroke-style, fill and transformations.

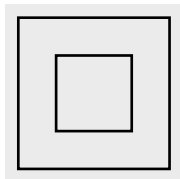`#group(content, name: none)`

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

3.5. Transformations

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

3.5.1. Translate

`#translate(coordinate)`

```
// Outer rect
rect((0,0), (2,2))
// Inner rect
translate((.5,.5,0))
rect((0,0), (1,1))
```
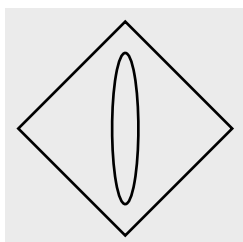
3.5.2. Set Origin

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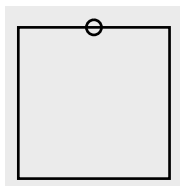
```
// Outer rect
rect((0,0), (2,2), name: "r")
// Move origin to top edge
set-origin("r.above")
circle((0, 0), radius: .1)
```

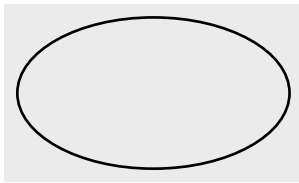### 3.5.3. Rotate

`#rotate(axis-dictionary)`
`#rotate(z-angle)`

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

3.5.4. Scale

```
#scale(axis-dictionary)
#scale(factor)
```

```
// Scale x-axis
scale((x: 1.8))
circle((0,0))
```

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> or <length>   (default: 0)
      The number of units in the x direction.

y <number> or <length>   (default: 0)
      The number of units in the y direction.

z <number> or <length>   (default: 0)
      The number of units in the z direction.

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

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *

  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).
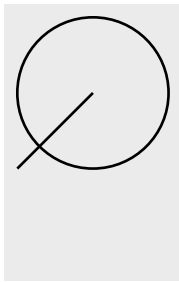
```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  line((0,0), (1, 1))

  // Draws a circle at (1,1)
  circle(())
})
```

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 <bool>   (default: true)
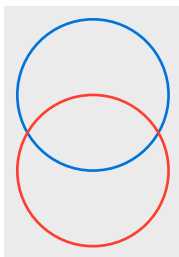 When false the previous position will not be updated.

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.

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  circle((0, 0), stroke: blue)
  circle((rel: (0, -1)), stroke: red)
})
```

4.4. Polar

Defines a point a `radius` distance away from the origin at the given `angle`. An angle of zero degrees. An angle of zero degrees is to the right, a degree of 90 is upward.

angle <angle>
  The angle of the coordinate.
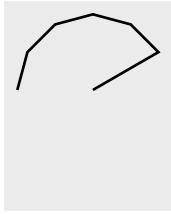
radius <number> or <length> or <array of length or number>
  The distance from the origin. An array can be given, in the form (x, y) to define the x and y radii of an ellipse instead of a circle.

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  line((0,0), (angle: 30deg, radius: 1cm))
})
```

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

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
    import cetz.draw: *
    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 ..., v_n and numbers α_1, α_2, ..., α_n. Then the barycentric coordinate specified by these vectors and numbers is

$$\frac{\alpha_1 v_1 + \alpha_2 v_1 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

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.
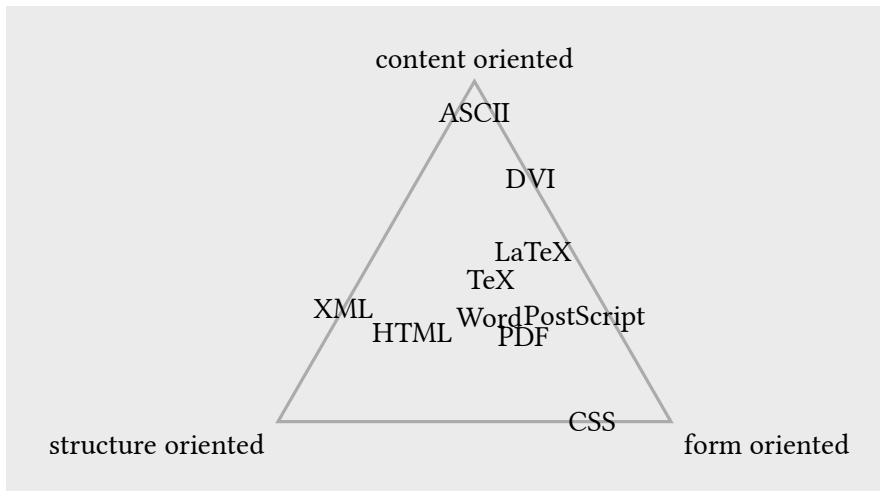
```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  circle((90deg, 3), radius: 0, name: "content")
  circle((210deg, 3), radius: 0, name: "structure")
  circle((-30deg, 3), radius: 0, name: "form")

  for (c, a) in (
    ("content", "bottom"),
    ("structure", "top-right"),
    ("form", "top-left")
  ) {
    content(c, box(c + " oriented", inset: 5pt), anchor: a)
  }

  stroke(gray + 1.2pt)
  line("content", "structure", "form", close: true)

  for (c, s, f, cont) in (
    (0.5, 0.1, 1, "PostScript"),
    (1, 0, 0.4, "DVI"),
    (0.5, 0.5, 1, "PDF"),
    (0, 0.25, 1, "CSS"),
    (0.5, 1, 0, "XML"),
    (0.5, 1, 0.4, "HTML"),
    (1, 0.2, 0.8, "LaTeX"),
    (1, 0.6, 0.8, "TeX"),
    (0.8, 0.8, 1, "Word"),
    (1, 0.05, 0.05, "ASCII")
  ) {
    content((bary: (content: c, structure: s, form: f)), cont)
  }
})
```

4.6. Anchor

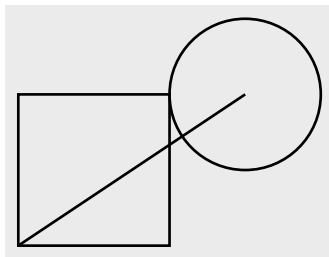Defines a point relative to a named element using anchors, see Section 2.2.

name <string>

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

anchor `<string>`

> An anchor of the element. If one is not given a default anchor will be used. On most elements this is `center` but it can be different.

You can also use implicit syntax of a dot separated string in the form `"name.anchor"`.

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  line((0,0), (3,2), name: "line")
  circle("line.end", name: "circle")
  rect("line.start", "circle.left")
})
```

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.

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  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 $)

  stroke(red)
  line(
    "a",
    (element: "c", point: "a", solution: 1),
    "c",
    (node: "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.
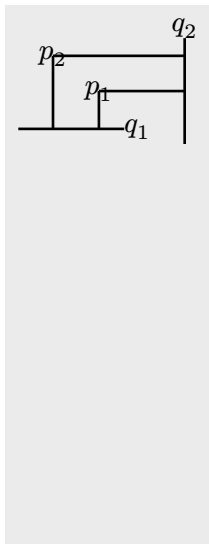
horizontal `<coordinate>`
> The coordinate through which the horizontal line passes.

vertical `<coordinate>`
> The coordinate through which the vertical line passes.

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

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  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: "left")
  line((2, -0.2), (2, 1.2), name: "yline")
  content("yline.end", $ q_2 $, anchor: "bottom")

  line("p1", (horizontal: (), vertical: "xline"))
  line("p2", (horizontal: (), vertical: "xline"))
  line("p1", (vertical: (), horizontal: "yline"))
  line("p2", (vertical: (), horizontal: "yline"))
})
```

4.9. Interpolation

Use this to linearly interpolate between two coordinates a and b with a given factor number. If number is a `<length>` the position will be at the given distance away from a towards 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 `<number>` or `<length>`
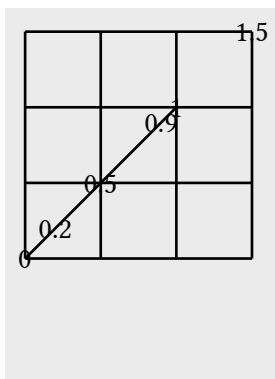> The factor to interpolate by or the distance away from a towards b.

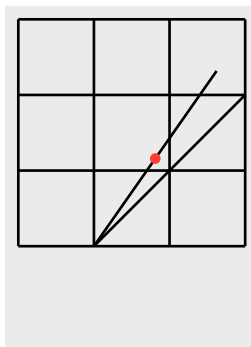**angle** `<angle>`   (default: `0deg`)

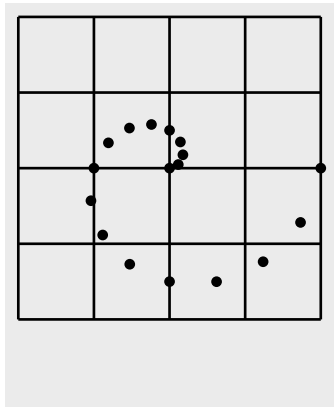Can be used implicitly as an array in the form (a, number, b) or (a, number, angle, b).

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  grid((0,0), (3,3), help-lines: true)

  line((0,0), (2,2))

  for i in (0, 0.2, 0.5, 0.9, 1, 1.5) {
    content(((0,0), i, (2,2)), [#i])
  }
})
```
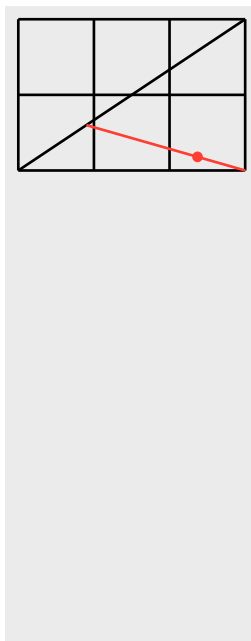
```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  grid((0,0), (3,3), help-lines: true)
  line((1,0), (3,2))
  line((1,0), ((1, 0), 1, 10deg, (3,2)))
  fill(red)
  stroke(none)
  circle(((1, 0), 0.5, 10deg, (3, 2)), radius: 2pt)}
})
```
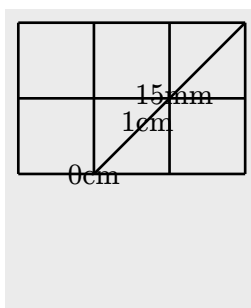
```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  grid((0,0), (4,4), help-lines: true)

  fill(black)
  stroke(none)
  let n = 16
  for i in range(0, n+1) {
    circle(((2,2), i / 8, i * 22.5deg, (3,2)), radius: 2pt)
  }
})
```

You can even chain them together!

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  grid((0,0), (3, 2), help-lines: true)
  line((0,0), (3,2))
  stroke(red)
  line(((0,0), 0.3, (3,2)), (3,0))
  fill(red)
  stroke(none)
  circle(
    (
      // a
      (((0, 0), 0.3, (3, 2))),
      0.7,
      (3,0)
    ),
    radius: 2pt
  )
})
```

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  grid((0,0), (3, 2), help-lines: true)
  line((1,0), (3,2))
  for (l, c) in ((0cm, "0cm"), (1cm, "1cm"), (15mm, "15mm")) {
    content(((1,0), l, (3,2)), $ #c $)
  }
})
```

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.

```
#import "@local/cetz:0.0.1"
#cetz.canvas({
  import cetz.draw: *
  circle((0, 0), name: "c")
  fill(red)
  circle((v => cetz.vector.add(v, (0, -1)), "c.right"), radius: 0.3)
})
```