LaTeX&friends course in Helsinki

a TeX/LaTeX enthusiast's view

Gaetano Zanghirati
University of Ferrara, Italy

in cooperation with DOMAST
Doctoral School in Mathematics and Statistics
University of Helsinki, Finland

Helsinki, May 2019

TikZ-PGF

- Introduction and general principles
- Coordinate systems
- Nodes
- Graphs
- Plots of functions
- Colors, patterns and transparences
- Lines, curves and surfaces

Introduction

According to Till Tantau:

- TikZ just defines a number of TeX commands that draw graphics

when you use TikZ you "program" your graphics, just as you "program" your document when you use TeX/LaTeX. Just as TeX/LaTeX provide a special notation for formulas, TikZ provides a special notation for graphics.

- TikZ *ist kein Zeichenprogramm*, *i.e.*, TikZ is not a drawing program
- PGF, "portable graphics format" for TeX, is the underlying "basic level" of the system
- you get all the advantages of the "TeX-approach to typesetting" for your graphics: quick creation of simple graphics, precise positioning, the use of macros, often superior typography
- you also inherit all the disadvantages: steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really "show" how things will look like.

General principles

A layered system

- **Top – TikZ frontend:** easy to use for humans, succinct, slow
- **Middle – PGF Basic layer:** TEX macros for creating figures, easy to use for other packages, verbose, quick.
  Provides a set of basic commands that allow to produce complex graphics in a much easier manner than by using the system layer directly.
  Constituted by: a core (includes several interdependent packages, can only be loaded at once) and additional modules (extend the core, more special-purpose commands like node management or plotting interface)
- **Bottom – PGF System layer:** Minimalistic set of TEX macros for creating figures, different implementation for each backend driver, extremely difficult to use, extremely fast (as fast as normal TEX).
  Provides a complete abstraction of what is going on "in the driver", like `dvips` or `dvipdfm` (from `.dvi` file to `.ps` or `.pdf` file). Each driver has its own syntax: PGF's system layer abstracts away these differences, by converting to different TEX's `\special` commands.

General principles

Example:

```
\tikz \draw (0,0) -- (30:10pt) -- (60:10pt) -- cycle;
```

General principles

Example:

```
\tikz \draw (0,0) -- (30:10pt) -- (60:10pt) -- cycle;
```

```
\pgfpathmoveto{\pgfpointxy{0}{0}}
\pgfpathlineto{\pgfpointpolar{30}{10pt}}
\pgfpathlineto{\pgfpointpolar{60}{10pt}}
\pgfpathclose
\pgfusepath{draw}
```

TikZ → PGF Basic layer

General principles

Example:

```
\tikz \draw (0,0) -- (30:10pt) -- (60:10pt) -- cycle;
```

```
\pgfpathmoveto{\pgfpointxy{0}{0}}
\pgfpathlineto{\pgfpointpolar{30}{10pt}}
\pgfpathlineto{\pgfpointpolar{60}{10pt}}
\pgfpathclose
\pgfusepath{draw}
```

TikZ → PGF Basic layer

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{8.660254pt}{5pt}
\pgfsys@lineto{5pt}{8.660254pt}
\pgfsys@closepath
\pgfsys@stroke
```

PGF Basic layer → PGF System layer

General principles

Example:

```
\tikz \draw (0,0) -- (30:10pt) -- (60:10pt) -- cycle;
```

```
\pgfpathmoveto{\pgfpointxy{0}{0}}
\pgfpathlineto{\pgfpointpolar{30}{10pt}}
\pgfpathlineto{\pgfpointpolar{60}{10pt}}
\pgfpathclose
\pgfusepath{draw}
```

TikZ → PGF Basic layer

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{8.660254pt}{5pt}
\pgfsys@lineto{5pt}{8.660254pt}
\pgfsys@closepath
\pgfsys@stroke
```

PGF Basic layer → PGF System layer

```
\special{pdf: 0 0 m}
\special{pdf: 8.627899 4.98132 l}
\special{pdf: 4.98132 8.627899 l}
\special{pdf: h}
\special{pdf: S}
```

PGF System layer → TEX `\special` for pdftex

General principles

TikZ: the PGF frontend

- in practice, the only "serious" frontend for PGF.

TikZ is a set of commands that gives access to all features of PGF, but makes using the basic layer much easier

- syntax: a mixture of METAFONT and PSTricks and some ideas of Till Tantau himself
- problem with directly using the basic layer: the code is often too "detailed". With the TikZ frontend a single simple METAFONT-like command can do the job of many low level commands
- other frontends exist, much less complete as alternatives to TikZ, such as the pgfpict2e package, that reimplements the standard LaTeX `picture` environment and commands like `\line` or `\vector` using the PGF basic layer (the `picture` environment is also reimplemented by the pict2e package without using PGF at all).
  But, well... **abandon** `picture`!!

Alternatives?

- standard LaTeX `picture` environment: very portable, but hugely limited compared to TikZ (little more than quite simple graphics)
- PSTricks package: powerful enough to create any conceivable kind of graphic, with many nice extra packages for special purposes, but it is not really portable and, most importantly, it does not work with `pdftex`.
- xypic package: an older package for creating graphics, harder to use and to learn because its slightly cryptic syntax and documentation
- dratex package: a very small graphic package, for not too complex tasks
- METAPOST program: a powerful alternative to TikZ. It used to be an external program, which entailed a bunch of problems, but in LuaTeX it is now build in. Inclusion of labels is much easier to achieve by using PGF.
- XFig program: a free WYSIWYG alternative to TikZ for users who do not wish to "program" their graphics. There is a program that will convert XFig graphics to TikZ.

General principles

In TikZ

- the basic command names and the notion of path operations are taken from METAFONT
- the option mechanism comes from PSTricks
- the notion of styles comes from SVG
- the graph syntax is taken from GRAPHVIZ
- the coordinate transformations are introduced by Till Tantau
- to make it all work together, some compromises are also accepted

```
\usepackage{tikz}
\usetikzlibrary{arrows,shapes,positioning,....} % loads extensions
...
\begin{tikzpicture} ... \end{tikzpicture}
\tikz ...  % for single inline commands
```

Design principles

Special syntax for specifying points and coordinates

- 2D cartesian coordinates: two TEX dimensions, separated by comma, in round brackets as in `(1cm,2pt)`. It is given in the PGF's *xy*-coordinate system
- 3D cartesian coordinates: three TEX dimensions, as in `(1cm,2pt,0.4em)`. It is given in the PGF's *xyz*-coordinate system
- polar coordinates: use a colon instead of a comma as in `(30:1cm)`, which means "1cm in a 30 degrees direction"
- anchor of a previously defined shape: `(first node.south)`
- relative coordinates: add a single "+" before the coordinate, as in `+(1cm,0pt)`, to mean an absolute displacement from a previous point/coordinate, which does not change. For ex., `(1,0) +(1,0) +(0,1)` specifies displ. from `(1,0)`, *i.e.*, the three coordinates `(1,0)`, then `(2,0)` and `(1,1)`
- moving relative coordinates: add two "+" sign in front of the coordinate, as in `++(1cm,0pt)`, to mean a relative displacement from the last point/coordinate used. For example, `(1,0) ++(1,0) ++(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(2,1)`.
- Default unit: 1cm.

Design principles

Special syntax for path specifications

- A path is a series of straight or curved lines, which need not be connected
- For example, to specify a triangular path:

```
\tikz \draw (10pt,0pt) -- (0pt,0pt) -- (0pt,10pt) -- cycle;
```

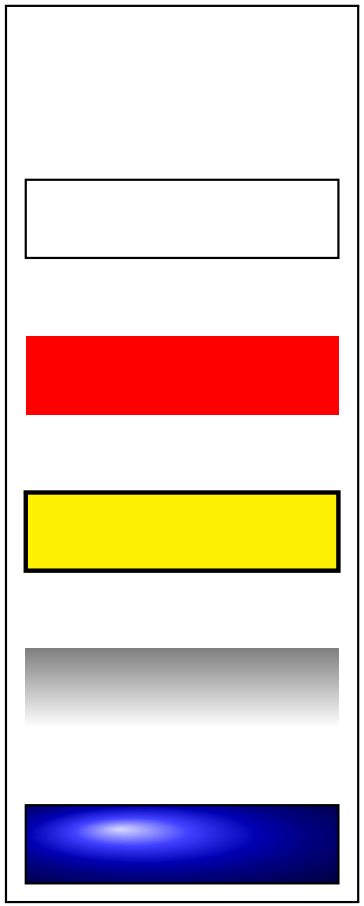
Actions on Paths

- path: a series of straight and curved lines. One can draw a path, fill a path, shade it, clip it, or do any combination of these.
- drawing (also known as stroking): can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas.
- filling: the interior of the path is filled with a uniform color. It makes sense only for closed paths. A path is automatically closed prior to filling, if necessary.
- shading: fill with a "pattern" the inner area of a closed path, instead of using a constant color
- clipping: clip the inner area of a closed path

TikZ allows to use different colors for filling and stroking

Design principles

Examples:



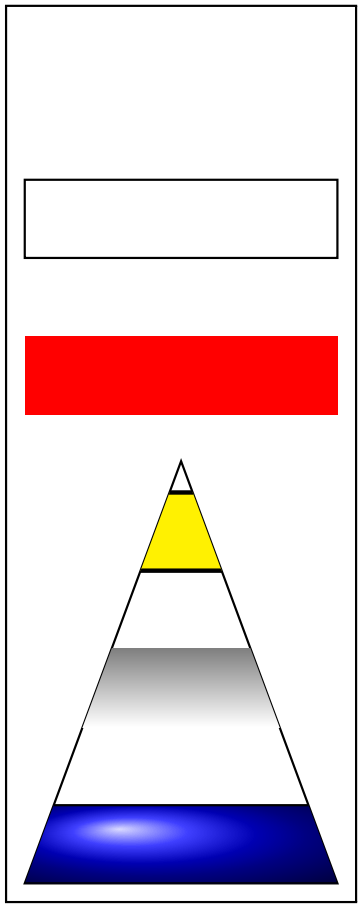
```
\begin{tikzpicture}
  \path (0,5) rectangle +(2,0.5); % see nothing!
  \draw (0,4) rectangle +(2,0.5);
  \fill[red] (0,3) rectangle +(2,0.5);

  \filldraw[fill=yellow] (0,2) rectangle +(2,0.5);
  \shade (0,1) rectangle +(2,0.5);
  \shadedraw[shading=ball] (0,0) rectangle +(2,0.5);
\end{tikzpicture}
```

Alias commands: `\draw` = `\path[draw]`, `\fill` = `\path[fill]`, `\filldraw` = `\path[fill,draw]`, `\shade` = `\path[shade]`, `\shadedraw` = `\path[shade,draw]`, `\clip` = `\path[clip]`

Design principles

Examples:



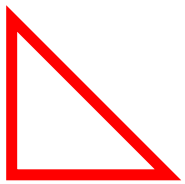
```
\begin{tikzpicture}
  \path (0,5) rectangle +(2,0.5); % see nothing!
  \draw (0,4) rectangle +(2,0.5);
  \fill[red] (0,3) rectangle +(2,0.5);
  \path[draw,clip](0,0) -- (2,0) -- (1,2.7) -- cycle;
  \filldraw[fill=yellow] (0,2) rectangle +(2,0.5);
  \shade (0,1) rectangle +(2,0.5);
  \shadedraw[shading=ball] (0,0) rectangle +(2,0.5);
\end{tikzpicture}
```

Alias commands: `\draw` = `\path[draw]`, `\fill` = `\path[fill]`, `\filldraw` = `\path[fill,draw]`, `\shade` = `\path[shade]`, `\shadedraw` = `\path[shade,draw]`, `\clip` = `\path[clip]`

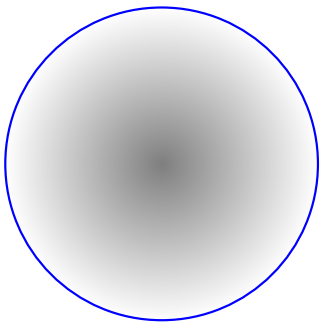
Design principles

Key-Value syntax for graphic parameters

- whenever TikZ draws or fills a path, a large number of graphic parameters influences the rendering. Examples: colors, dashing patterns, clipping area, line width, and many others.
- all these options are specified as lists of so called **key-value pairs**, that are passed as optional parameters to the path drawing and filling commands



```
\tikz \draw[line width=2pt,color=red] %
            (1,0) -- (0,0) -- (0,1) -- cycle;
```

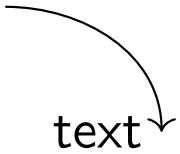


```
\tikz \shadedraw[draw=blue,shading=radial] %
                  (0,0) circle (1);
```

Design principles

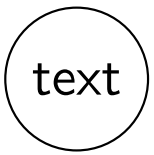
Special syntax for specifying nodes

- node: an element (container) to be added to the graphic
- they can contain text, other nodes, pictures, ...
- can be added to a path




```
\tikz \draw[<-] (1,0) node[anchor=east] {text} %
            to[out=90,in=0] (0,0.8);
```

- nodes are inserted at the current position of the path, but either after (the default) or before the complete path is rendered
- if text options are given:



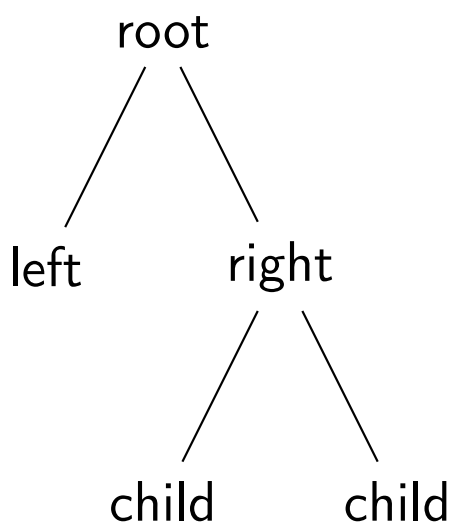

```
\tikz \draw (1,0) node[circle,draw] {text};
```

  the text is not just put at the current position, but options are applied to the node as an "object".
- a name ca be added to a node for later reference, either by using the option `name=nodeName` or by stating the node name in parentheses outside the text: as in `\node[circle] (name) {text};`.
- predefined shapes include `rectangle`, `circle` and `ellipse`. Additional are available thorough the shapes library and it is also possible to define new ones.

Design principles

Special syntax for specifying trees

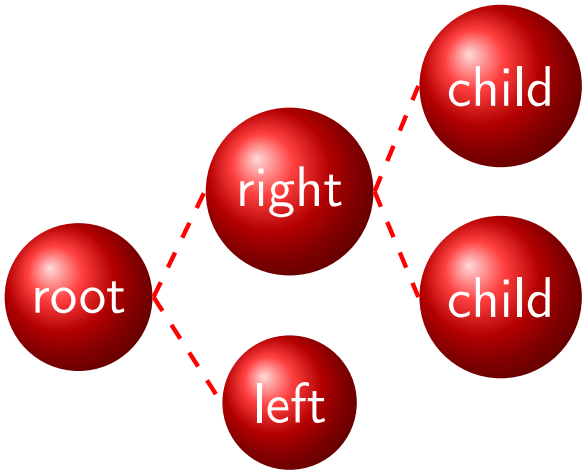
- the node syntax can also be used to draw trees
- in a tree, a node can be followed by any number of children, each introduced by the keyword `child`
- children are nodes themselves, each of which may have children in turn




```
\begin{tikzpicture}
  \node {root} %
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Design principles

– it is possible to use options to modify the way trees are drawn



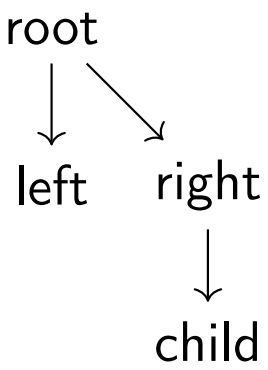

```
\begin{tikzpicture}[scale=0.9,    %
   parent anchor=east,            %
   child anchor=west, grow=east,  %
   sibling distance=15mm,         %
   level distance=15mm,           %
   every node/.style=             %
     {ball color=red,circle,text=white},%
   edge from parent/.style=       %
                  {draw,dashed,thick,red}]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```

*Comment out end-of-line after commas in options lists is just a good practice, it's not mandatory

Design principles

Special syntax for graphs

- graph syntax: another syntax layer build "on top" of the node syntax
- the \graph command extends the so-called dot-notation used in the popular GRAPHVIZ program



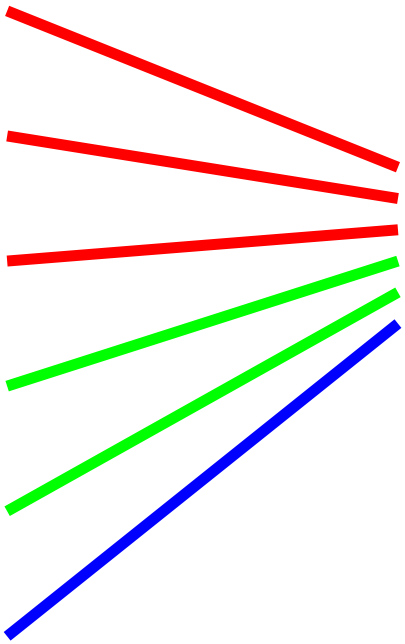

```
\usetikzlibrary{graphs}
\tikz \graph[grow down, branch right] {
  root -> {left, right -> {child, child}}
};
```

- if the compiler allows to call Lua code (as luatex does), one can ask TikZ to automatically compute good positions for the nodes of a graph using one of several integrated graph drawing algorithms
- \graph command: useful when there are numerous fairly similar nodes that only differ with respect to the name they show

Design principles

Grouping of graphic parameters

- graphic parameters common to several path drawing or filling commands cam be grouped as optional parameter of a `scope` environment
- Encapsulation: they will apply only to the drawing and filling commands inside the environment
- nested `scope` environments or individual drawing commands can override the graphic parameters of outer `scope` environments
- The `tikzpicture` environment itself also behaves like a `scope` environment



```
\begin{tikzpicture}[line width=2pt]
\begin{scope}[color=red]
  \draw (0, 2)   -- (2.5, 1)
        (0, 1.2) -- (2.5, 0.8)
        (0, 0.4) -- (2.5, 0.6);
\end{scope} \begin{scope}[color=green]
\draw (0, -0.4) -- (2.5, 0.4);
\draw (0, -1.2) -- (2.5, 0.2);
\draw[color=blue] (0, -2) -- (2.5, 0);
\end{scope} \end{tikzpicture}
```

Design principles

Coordinate transformation system

- TikZ supports both PGF's coordinate transformation systems
- performs transformations as well as canvas transformations, a more low-level transformation system
- it is made deliberately harder to use canvas transformations than coordinate transformations
- coordinate transformation: it's the native PGF system
- canvas transformation: it is a PostScript operation
  - the canvas transformation must be used with great care and often results in "bad" graphics, with changing line width and text in wrong sizes
  - PGF loses track of where nodes and shapes are positioned when canvas transformations are used.

  Example: scaling by 3 in the x-dir and by 0.5 in the y-dir. Canvas: everything is scaled by these same factors (including the thickness of lines and text). Coordinate system: only coordinates are scaled, but not line width nor text:

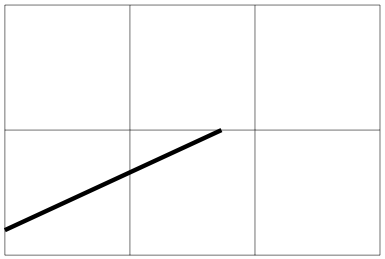
- default: all transformations only apply to the coordinate transformation system
- `\pgflowlevel` allows to apply a canvas transformation

Coordinate systems

- coordinate: a position on the canvas where the picture is drawn
- coordinates are always put in round brackets
- syntax: `([options]⟨coordSpecs⟩)`
- ⟨`coordSpecs`⟩: can be given in one of many different possible coordinate systems:
  - Cartesian
  - affine
  - polar 2D
  - isometric 3D
  - barycentric
  - user defined
- no matter which coordinate system is used, in the end, a specific point on the canvas is represented by the coordinate

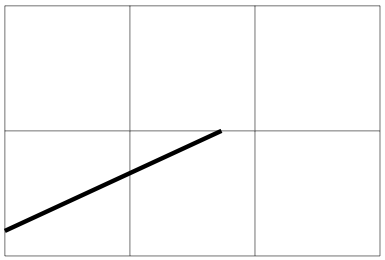
Coordinate systems

Explicit: by using the keyword "cs:" (for "coordinate system") and the syntax: (⟨coordSys⟩ cs: ⟨listOfCoords⟩)
where ⟨listOfCoords⟩ is a list of key-value pairs specific to the selected coordinate system.
Explicit specification is often too verbose



```
\begin{tikzpicture}[scale=0.8]
  \draw[help lines] (0,0) grid (3,2);
  \draw[thick] (canvas cs: x=0cm, y=2mm) %
     -- (canvas polar cs: radius=2cm, angle=30);
\end{tikzpicture}
```

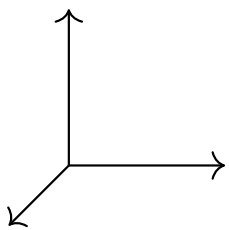
Implicit: a special syntax is provided for the coordinate systems that are likely to be used more often. TikZ recognizes the special syntax and automatically selects the correct coordinate system



```
\begin{tikzpicture}[scale=0.8]
  \draw[help lines] (0,0) grid (3,2);
  \draw[thick] (0cm,2mm) -- (30:2cm);
\end{tikzpicture}
```

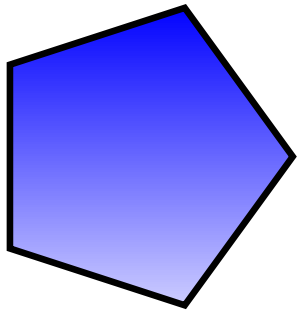
Coordinate systems

Implicit 3D Cartesian coordinate system:



```
\begin{tikzpicture}
\draw [->] (0,0,0) -- (1,0,0);
\draw [->] (0,0,0) -- (0,1,0);
\draw [->] (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

Implicit 2D polar coordinate system:



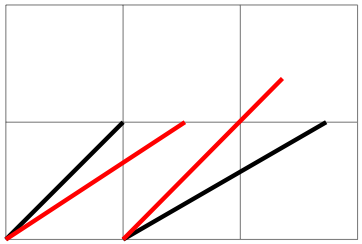
```
\begin{tikzpicture}
\draw [top color=blue,bottom color=blue!20,%
       draw,very thick] %
       (0:1cm) -- (72:1cm) -- (144:1cm) %
       -- (216:1cm) -- (288:1cm) -- cycle;
\end{tikzpicture}
```

Coordinate systems

It is possible to give options that apply only to a single coordinate, although this makes sense for transformation options only.

To give transformation options for a single coordinate, give these options at the beginning in brackets

Example:



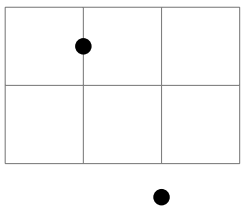
```
\begin{tikzpicture}[scale=0.75,thick]
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (1,1);
  \draw[red] (0,0) -- ([xshift=15pt] 1,1);
  \draw      (1,0) -- +(30:2cm);
  \draw[red] (1,0) -- +([shift=(135:15pt)] 30:2cm);
\end{tikzpicture}
```

Coordinate systems

Coordinate system `canvas`: a dimension d_x using the `x=` option and another dimension d_y using the `y=` option.
`x=`⟨`dimension`⟩ (no default, initially 0pt) distance on the right of the origin.
`y=`⟨`dimension`⟩ (no default, initially 0pt) distance above the origin.

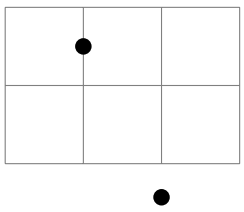
negative values reverse the direction, as usual

Explicit version:



```
\begin{tikzpicture}[scale=0.5]
  \draw[help lines] (0,0) grid (3,2);
  \fill (canvas cs:x=1cm,y=1.5cm)    circle (3pt);
  \fill (canvas cs:x=2cm,y=-5mm+2pt) circle (3pt);
\end{tikzpicture}
```

Implicit version:



```
\begin{tikzpicture}[scale=0.5]
  \draw[help lines] (0,0) grid (3,2);
  \fill (1cm,1.5cm)    circle (3pt);
  \fill (2cm,-5mm+2pt) circle (3pt);
\end{tikzpicture}
```

Dimensions like `1cm+2pt` are legal: the mathematical engine is used for evaluation

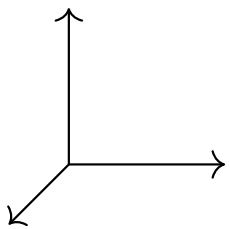
Coordinate systems

Coordinate systems `xy` and `xyz`: specify a point as a linear combination of the three vectors called the *x*-, *y*-, and *z*-vectors.
Default: the *x*-vector points 1cm to the right, the *y*-vector points 1cm upwards, the *z*-vector points to `(-3.85mm, -3.85mm)`

The *x*- and *y*-vectors can be changed arbitrarily, while *z*-vector has **fixed direction**

`xyz` **is not** a complete 3D coordinate system! It is more an axonometric-like projection. At the moment, the only way to work with a **fully 3D** Cartesian coordinate system is by using the tikz-3dplot package

Syntax: `x=`⟨`factor`⟩, `y=`⟨`factor`⟩, `z=`⟨`factor`⟩.



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (xyz cs:x=1);
  \draw (0,0) -- (xyz cs:y=1);
  \draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

Coordinate systems

Inference rules for implicit coordinates

Coordinates like `(1,2cm)` are neither canvas coordinates nor `xyz` coordinates:

- if a coordinate is of the fully implicit form `(⟨x⟩,⟨y⟩)`, then `⟨x⟩` and `⟨y⟩` are checked, independently:
  - both have a dimension: the `canvas` coordinate system is used
  - both lack a dimension: the `xyz` coordinate system is used
  - `⟨x⟩` has a dimension and `⟨y⟩` has not: `(⟨x⟩,0pt) + (0,⟨y⟩)`
  - `⟨y⟩` has a dimension and `⟨x⟩` has not: `(⟨x⟩,0) + (0pt,⟨y⟩)`
- pay attention to mixing dimensionless values and dimensions: `(2+3cm,0)`≠`(2cm+3cm,0)` !! Instead, `(2+3cm,0)`=`(2pt+3cm,0)`

If values are mixed, all dimensionless values are "upgraded" to `pt` dimension!

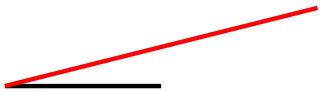
Coordinate systems

To change the *x*-, *y*-, and *z*-vectors, use **options to the path-building command** (not to the coordinate). Syntax: `x=`⟨`value`⟩

- if ⟨`value`⟩ is a dimension, the *x*-vector of PGF's `xyz`-coordinate system is set up to point to (⟨`value`⟩, `0pt`)
- if ⟨`value`⟩ is a coordinate, the *x*-vector of PGF's `xyz`-coordinate system is set up to point to the specified coordinate

```
\begin{tikzpicture}[thick]
\draw (0,0) -- +(1,0);
\draw[x=2cm,color=red] (0,0.2) -- +(1,0);
\end{tikzpicture}
```

If ⟨`value`⟩ contains a comma, it must be put in braces

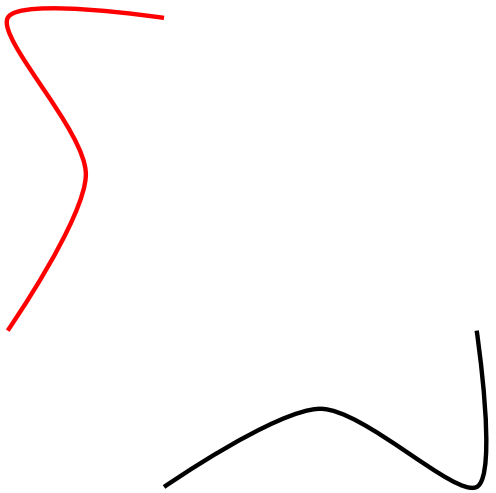


```
\begin{tikzpicture}[thick]
\draw (0,0) -- (1,0);
\draw[x={(2cm,0.5cm)},color=red] (0,0) -- (1,0);
\end{tikzpicture}
```

The size of steppings in grids are not affected by the *x*-vector

Coordinate systems

Example: exchange the meaning of the x- and y-coordinate



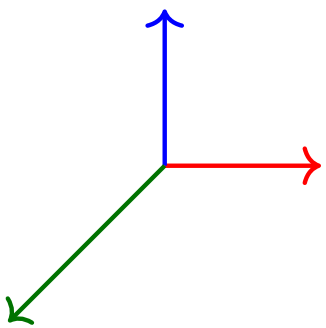
```
\begin{tikzpicture}[smooth,thick]
\draw[thick] plot coordinates%
      {(1,0) (2,0.5) (3,0) (3,1)};
\draw[x={(0cm,1cm)},y={(1cm,0cm)},color=red,%
      thick] plot coordinates%
      {(1,0) (2,0.5) (3,0) (3,1)};
\end{tikzpicture}
```

Syntax: `y=`⟨`value`⟩ (like the `x=` option)

if ⟨`value`⟩ is a dimension, the y-vector points to (0, ⟨`value`⟩)

Syntax: `z=`⟨`value`⟩ (like the `x=` and `y=` options)

if ⟨`value`⟩ is a dimension, gives always the point (⟨`value`⟩,⟨`value`⟩)

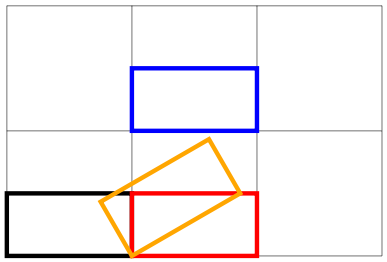


```
\begin{tikzpicture}[z=-1cm,->,thick]
  \draw[color=red] (0,0,0) -- (1,0,0);
  \draw[color=blue] (0,0,0) -- (0,1,0);
  \draw[color=darkgreen] (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

Coordinate systems

PGF and TikZ allow **coordinate transformations**

- each coordinate is first "reduced" to a position of the Cartesian plane with canonical *x*- and *y*-axes
- the next step is to apply the current coordinate transformation matrix: in general, any **affine transformation** (like translation, rotation, slanting, or scaling or any combination thereof) is possible
- Internally, PGF keeps track of a coordinate transformation matrix very much like the concatenation matrix used by PDF or PostScript)



```
\begin{tikzpicture}[thick,scale=0.8]
 \draw[help lines] (0,0) grid (3,2);
 \draw (0,0) rectangle (1,0.5);
 \begin{scope}[xshift=1cm]
  \draw [red] (0,0) rectangle (1,0.5);
  \draw[yshift=1cm,blue] (0,0) rectangle (1,0.5);
  \draw[rotate=30,orange] (0,0) rectangle (1,0.5);
 \end{scope}
\end{tikzpicture}
```

Coordinate systems

The coordinate transformation matrix applies to coordinates only

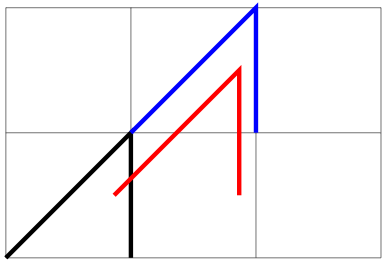
In particular, the coordinate transformation has no effect on things like the line width or the dash pattern or the shading angle

- general rule: if there is no 'coordinate' involved, even 'indirectly', the matrix is not applied
- sometimes, you simply have to try or look it up in the documentation
- the matrix cannot be set directly: you can only add another transformation to the current matrix
- all transformations are local to the current TEX group
- all transformations are added using graphic options,
- transformations apply immediately when they are encountered "in the middle of a path" and they apply only to the coordinates on the path following the transformation option

```
\tikz \draw[thick,red] (0,0) rectangle (0.5,0.3) %
                [xshift=1cm] (0,0) rectangle (0.5,0.3);
```

Coordinate systems

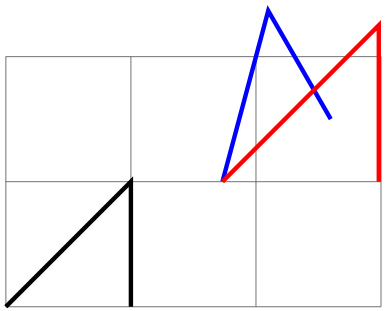
Option: shift=⟨coord⟩



```
\begin{tikzpicture}[thick,scale=0.8]
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[shift={(1,1)},blue](0,0) -- (1,1) -- (1,0);
\draw[shift={(30:1cm)},red]%
      (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

Option: shift only

This option does not take any parameter. Its effect is to cancel all current transformations except for the shifting.

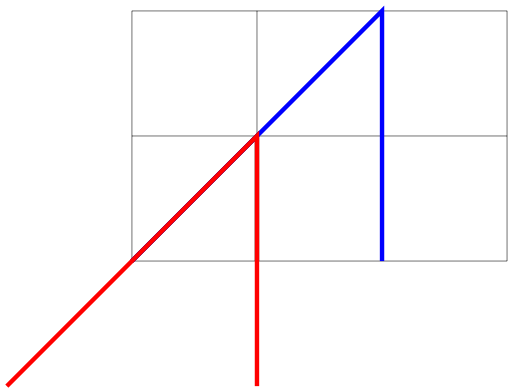


```
\begin{tikzpicture}[thick,scale=0.8]
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[rotate=30,xshift=2cm,blue] (0,0)--(1,1)--(1,0
\draw[rotate=30,xshift=2cm,shift only,red] %
      (0,0) -- (1,1) -- (1,0); \end{tikzpicture}
```

Coordinate systems

Options:

xshift=⟨dimen⟩	xscale=⟨factor⟩
yshift=⟨dimen⟩	yscale=⟨factor⟩
scale=⟨factor⟩	xslant=⟨factor⟩
scale around=⟨factor⟩:⟨coord⟩	yslant=⟨factor⟩



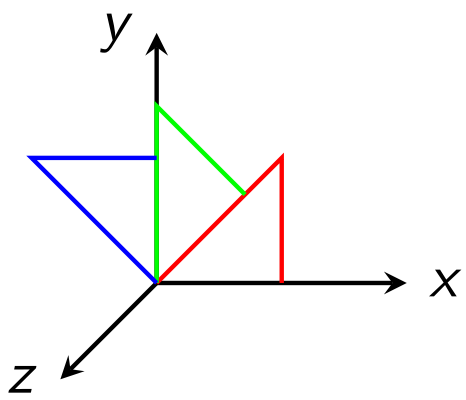
```
\begin{tikzpicture}[thick,scale=0.8]
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[scale=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[scale around={2:(1,1)},red] %
      (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

Coordinate systems

Options: rotate=⟨degrees⟩
rotate around x=⟨angle⟩
rotate around y=⟨angle⟩
rotate around z=⟨angle⟩

rotate around={⟨degrees⟩:⟨coord⟩}

positive angles result in an anticlockwise rotation

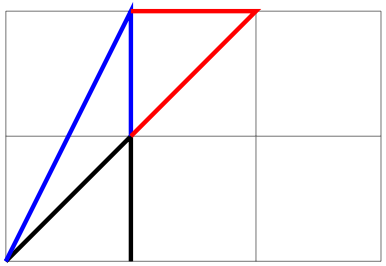



```
\begin{tikzpicture}[>=stealth,thick,scale=0.8]
\draw [->] (0,0,0) -- (2,0,0) %
           node [at end, right] {$x$};
\draw [->] (0,0,0) -- (0,2,0) %
           node [at end, left]  {$y$};
\draw [->] (0,0,0) -- (0,0,2) %
           node [at end, left]  {$z$};
\draw [red, rotate around z=0] %
           (0,0) -- (1,1) -- (1,0);
\draw [green, rotate around z=45] %
           (0,0) -- (1,1) -- (1,0);
\draw [blue, rotate around z=90]  %
           (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

Coordinate systems

Options for **affine transformation**: `cm={⟨a⟩,⟨b⟩,⟨c⟩,⟨d⟩,⟨coord⟩}`
let ⟨coord⟩ specify the point `(t,u)`. The option applies the affine transformation to all coordinates `(x,y)`

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t \\ u \end{pmatrix}$$



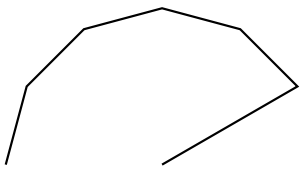
```
\begin{tikzpicture}[thick,scale=0.8]
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[cm={1,1,0,1,(0,0)},blue] %
      (0,0) -- (1,1) -- (1,0);
\draw[cm={0,1,1,0,(1cm,1cm)},red] %
      (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

Option `reset cm` completely resets the coordinate transformation matrix to the identity matrix. This will destroy not only the transformations applied in the current scope, but also all transformations inherited from surrounding scopes.

Coordinate systems

Coordinate system `canvas polar`
Allows to specify (planar) polar coordinates for a point on the canvas.
Options:
`angle=`⟨`degrees`⟩ with $-360 \leq \langle \texttt{degrees} \rangle \leq 720$
`radius=`⟨`dimension`⟩
`x radius=`⟨`dimension`⟩ `y radius=`⟨`dimension`⟩
Two different radii are specified by `(30:1cm and 2cm)`

```
\tikz \draw (0,0) -- (canvas polar cs:angle=30,radius=1cm);
```

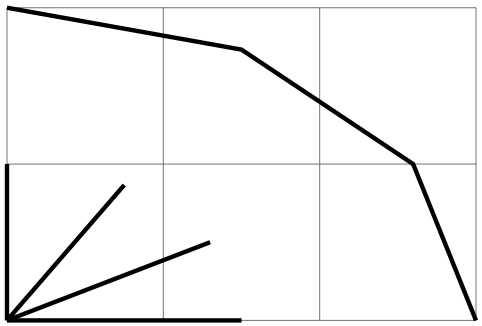


```
\tikz \draw  (0cm,0cm) -- (30:1cm) -- (60:1cm) %
  -- (90:1cm) -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

Special angles can be given as words (with obvious meanings): `up` (90°), `down`, `left`, `right`, `north`, `south`, `west`, `east`, `north east` (45°), `north west`, `south east`, `south west`.

Coordinate systems

Coordinate system `xyz polar`
Similar to the canvas polar system, but `radius` and `angle` are interpreted in the `xy`-coordinate system, not in the canvas system.
Options: same as `canvas polar`
The position of a point is specified by considering the ellipse whose half axes are the current *x*- and *y*-vectors, then considering the point that lies at a given angle on this ellipse (counterclockwise), and finally multiplying the resulting vector by the given radius ⟨`factor`⟩



```
\begin{tikzpicture}[x=1.5cm,y=1cm,thick]
\draw[help lines] (0cm,0cm) grid (3cm,2cm);
\draw (0,0) -- (xyz polar cs:angle=0,radius=1);
\draw (0,0) -- (xyz polar cs:angle=30,radius=1);
\draw (0,0) -- (xyz polar cs:angle=60,radius=1);
\draw (0,0) -- (xyz polar cs:angle=90,radius=1);
\draw (xyz polar cs:angle=0,radius=2)
   -- (xyz polar cs:angle=30,radius=2)
   -- (xyz polar cs:angle=60,radius=2)
   -- (xyz polar cs:angle=90,radius=2);
\end{tikzpicture}
```

Implicit form:

```
\tikz[x={(1.5cm,0cm)},y={(0cm,1cm)}]\draw (0,2) -- (30:2) -- (60:2) -- (90:2);
```

Coordinate systems

Coordinate system `xy polar`
Just an alias for `xyz polar`

Coordinate system `barycentric`
A point is expressed as the linear combination of multiple vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\ldots$, $\boldsymbol{v}_n$:

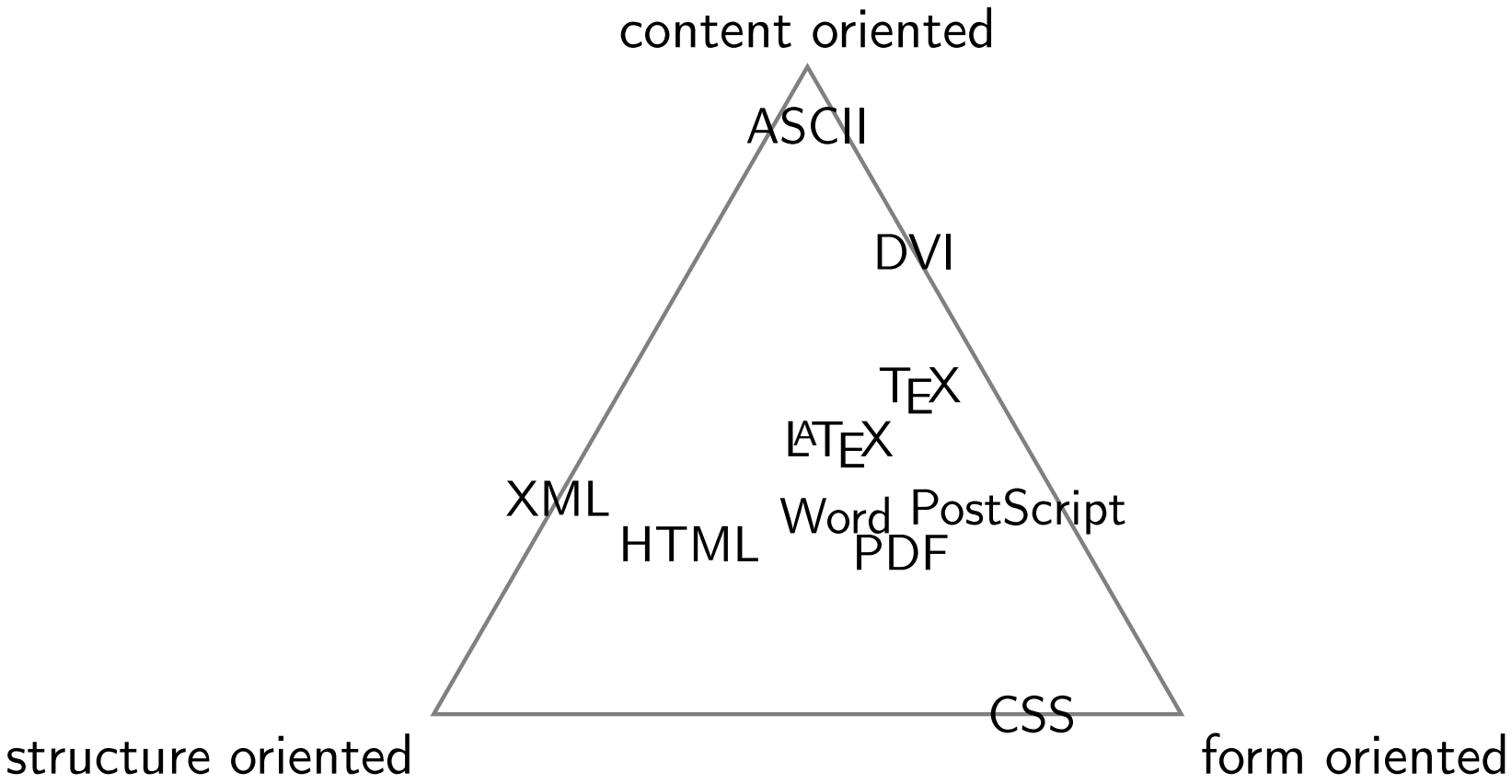
$$\frac{\alpha_1 \boldsymbol{v}_1 + \alpha_2 \boldsymbol{v}_2 + \ldots + \alpha_n \boldsymbol{v}_n}{\alpha_1 + \alpha_2 + \ldots + \alpha_n} \qquad \text{with } \alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{R}$$

⟨`coordSPecs`⟩: a comma-separated list of expressions of the form ⟨`nodeName`⟩=⟨`number`⟩.
The list should not contain any spaces before or after the ⟨`nodeName`⟩ (unlike normal key-value pairs).

The vector is the `center` anchor of the ⟨nodeName⟩. To specify another anchor, say, the `north` anchor of a node, create a new coordinate at this anchor using for instance `\coordinate (mynorth) at (mynode.north)`

Coordinate systems

Example (TikZ manual): a barycentric scheme for languages



Coordinate systems

```
\begin{tikzpicture}
  \coordinate (content)   at (90:3cm);
  \coordinate (structure) at (210:3cm);
  \coordinate (form)      at (-30:3cm);
  \node [above]       at (content)   {content oriented};
  \node [below left]  at (structure) {structure oriented};
  \node [below right] at (form)      {form oriented};
  \draw [thick,gray] (content.south) -- (structure.north east) %
                     -- (form.north west) -- cycle;
  \small
  \node at (barycentric cs:content=0.5,structure=0.1 ,form=1)    {PostScript};
  \node at (barycentric cs:content=1  ,structure=0   ,form=0.4)  {DVI};
  \node at (barycentric cs:content=0.5,structure=0.5 ,form=1)    {PDF};
  \node at (barycentric cs:content=0  ,structure=0.25,form=1)    {CSS};
  \node at (barycentric cs:content=0.5,structure=1   ,form=0)    {XML};
  \node at (barycentric cs:content=0.5,structure=1   ,form=0.4)  {HTML};
  \node at (barycentric cs:content=1  ,structure=0.2 ,form=0.8)  {\TeX};
  \node at (barycentric cs:content=1  ,structure=0.6 ,form=0.8)  {\LaTeX};
  \node at (barycentric cs:content=0.8,structure=0.8 ,form=1)    {Word};
  \node at (barycentric cs:content=1  ,structure=0.05,form=0.05) {ASCII};
\end{tikzpicture}
```

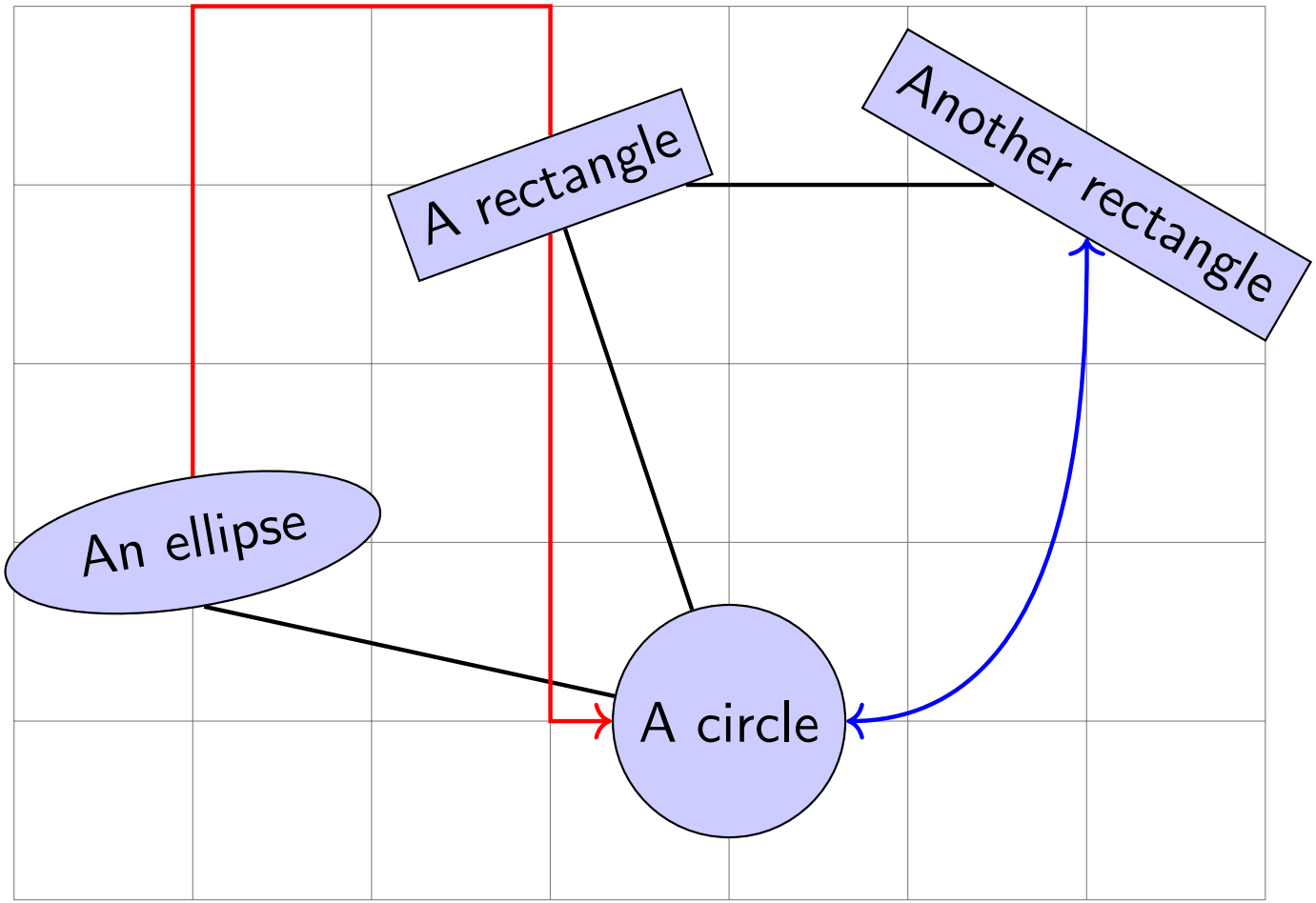
Coordinate systems

Coordinate system `node`
This coordinate system is used to reference a specific point inside or on the border of a previously defined node.
Options to specify which coordinate you mean:
`name=`⟨`nodeName`⟩
`anchor=`⟨`anchor`⟩
`angle=`⟨`degrees`⟩: this coordinate refers to a point of the node's border where a ray shot from the center in the given angle hits the border

- TikZ will be reasonably clever at determining the border points that you "mean", but this may fail in some situations: in these cases, the center will be used instead.
- The implicit way of specifying the `node` coordinate system is to simply use the name of the node in parentheses as in `(a)` or to specify a name together with an anchor or an angle separated by a dot as in `(a.north)` or `(a.10)`

Coordinate systems

Example:



Coordinate systems

```
\begin{tikzpicture}[fill=blue!20,thick]
  \draw[help lines] (-1,-2) grid (6,3);
  \path (0,0)  node(a) [ellipse,rotate=10,draw,fill] {An ellipse}
        (3,-1) node(b) [circle,draw,fill] {A circle}
        (2,2)  node(c) [rectangle,rotate=20,draw,fill] {A rectangle}
        (5,2)  node(d) [rectangle,rotate=-30,draw,fill] %
                                                {Another rectangle};
  \draw (a.south) -- (b) -- (c) -- (d);
  \draw[red,->] (a) |- +(1,3) -| (c) |- (b);
  \draw[blue,<->] (b) .. controls +(right:2cm) and +(down:1cm) .. (d);
\end{tikzpicture}
```

Coordinate systems

Coordinate system `tangent`
allows to compute the point that lies tangent to a shape

available only when the TikZ library calc is loaded

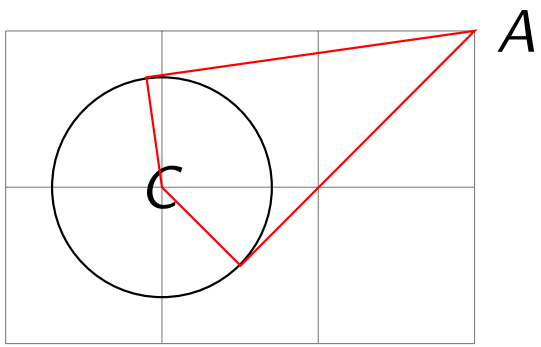
Options:
`node=`⟨`nodeName`⟩: the node on whose border the tangent should lie
`point=`⟨`point`⟩: the point through which the tangent should go
`solution=`⟨`number`⟩: if there are more than one solutions, specifies which one should be used

Currently, tangents can be computed only for nodes whose shape is `coordinate` or `circle`

Notice: there is no implicit syntax for this coordinate system

Coordinate systems

Example:

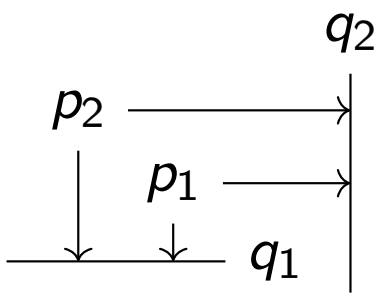



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \node[anchor=west] at (a) {$A$};
  \coordinate (a) at (3,2);
  \node [circle,draw] (c) at (1,1) [minimum size=40pt] {$C$};
  \draw[red] (a)  -- (tangent cs:node=c,point={(a)},solution=1) --
       (c.center) -- (tangent cs:node=c,point={(a)},solution=2) -- cycle;
\end{tikzpicture}
```

Coordinate systems

Coordinate system `perpendicular`
Options:
`horizontal line through=`⟨`coord`⟩: specifies that one line is a horizontal line that goes through the given coordinate
`vertical line through=`⟨`coord`⟩: specifies that the other line is vertical and goes through the given coordinate

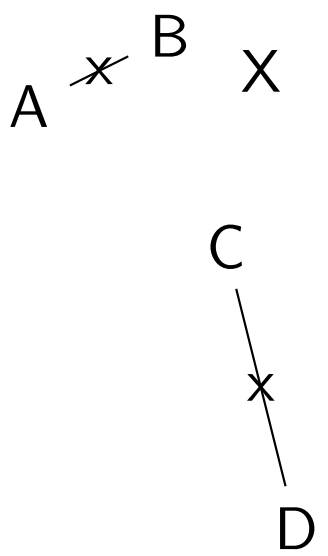
implicit syntax: `(`⟨p⟩ `|-` ⟨q⟩`)` or `(`⟨q⟩ `-|` ⟨p⟩`)`
Example: `(2,1 |- 3,4)` and `(3,4 -| 2,1)` both yield the same as (2,4) (provided the *xy*-coordinate system has not been modified).




```
\begin{tikzpicture}
  \path (30:1cm) node(p1) {$p_1$} (75:1cm) node(p2) {$p_2$};
  \draw (-0.2,0) -- (1.2,0) node(xline)[right] {$q_1$};
  \draw (2,-0.2) -- (2,1.2) node(yline)[above] {$q_2$};
  \draw[->] (p1) -- (p1 |- xline);
  \draw[->] (p2) -- (p2 |- xline);
  \draw[->] (p1) -- (p1 -| yline);
  \draw[->] (p2) -- (p2 -| yline);
\end{tikzpicture}
```

Coordinate systems

Note: in (⟨c⟩ |- ⟨d⟩) the coordinates ⟨c⟩ and ⟨d⟩ are not surrounded by parentheses. If complicated expressions are needed (like a computation using the $-syntax), surround them with braces: parentheses will then be added by TikZ around them.




```
\begin{tikzpicture}[scale=0.9]
  \node (A) at (0  ,1)   {A};
  \node (B) at (1  ,1.5) {B};
  \node (C) at (1.4,0)   {C};
  \node (D) at (1.9,-2)  {D};
  \draw (A) -- (B) node [midway] {x};
  \draw (C) -- (D) node [midway] {x};
  \node at ({$(A)!.5!(B)$} -| {$(C)!.5!(D)$}) {X};
\end{tikzpicture}
```

Coordinate systems: intersections

Intersections of **arbitrary paths**

```
\usetikzlibrary{intersections}
```

This library enables the calculation of intersections of two arbitrary paths. Due to the low accuracy of TEX, the paths should not be "too complicated". In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths.

To look for intersections of two paths, they must be **named** with the options
`name path=`⟨`name`⟩: this association survives beyond the final semi-colon of the path, but not the end of the surrounding scope
`name path global=`⟨`name`⟩: the association will survive beyond any scope

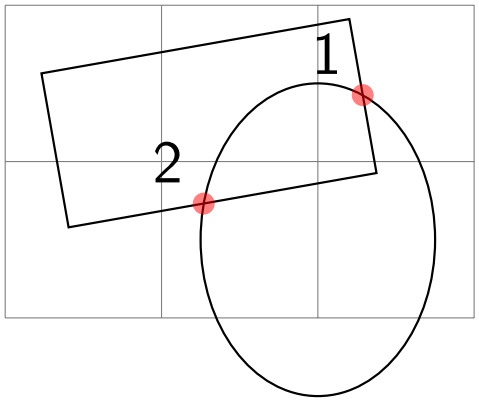
The actual intersection(s) of two named paths are obtained (if it/they exist(s)), by the following key:
`name intersections={opts}`

Coordinate systems: intersections

`opts`: keywords that determine, among other things, which paths to use for the intersection. Having processed the options, any intersections are then found. A coordinate is created at each intersection. By default, they are named `intersection-1`, `intersection-2`, ...
Optionally, the prefix "`intersection`" can be changed and the total number of intersections stored in a TEX macro

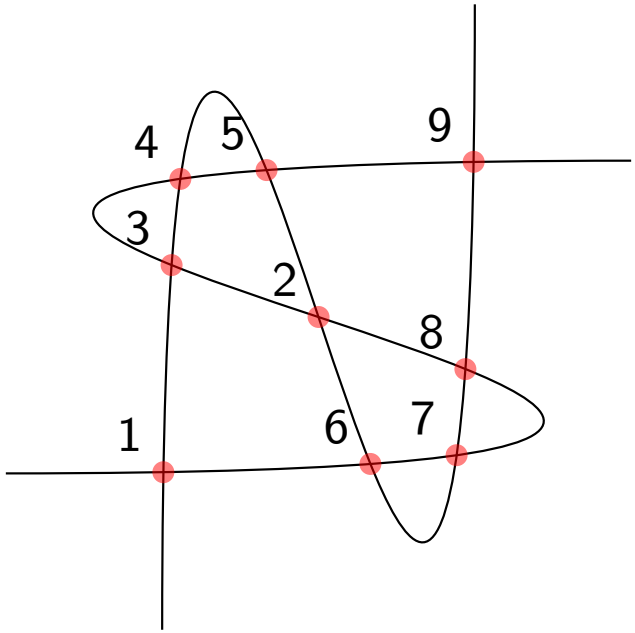
Inside `opts`, the following keys can be used:
`of=`⟨`namePath1`⟩ and ⟨`namePath2`⟩
`name=`⟨`prefix`⟩
`total=`⟨`macro`⟩
`sort by=`⟨`pathName`⟩: where `pathName` is one of the two paths
`by={`⟨`comma-separated list`⟩`}`: additional names for intersections. They are associated in the order they appear in the list. In case an element of the list starts with options in square brackets, these options are used when the coordinate is created. A coordinate name can still, but need not, follow the options.

Coordinate systems: intersections



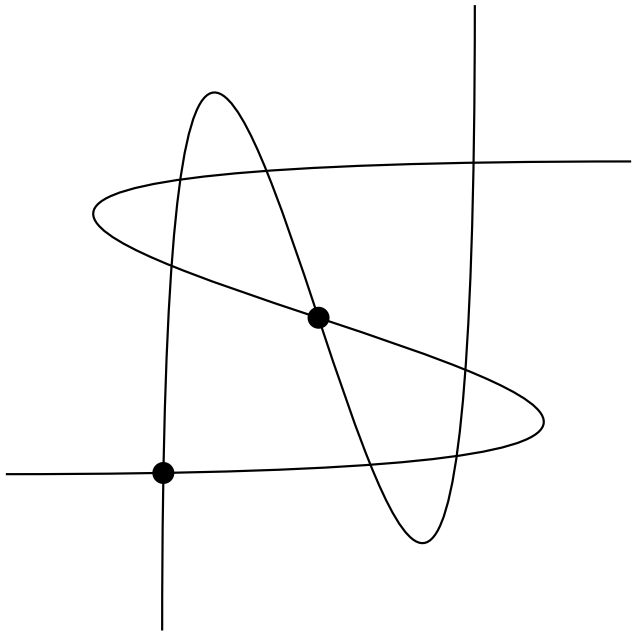

```
\begin{tikzpicture}%
  [every node/.style={opacity=1, black, above left}]
  \draw [help lines] grid (3,2);
  \draw [name path=ellipse] (2,0.5) ellipse (0.75cm and 1cm);
  \draw [name path=rectangle, rotate=10] %
        (0.5,0.5) rectangle +(2,1);
  \fill [red, opacity=0.5, %
        name intersections={of=ellipse and rectangle}]
    (intersection-1) circle (2pt) node {1}
    (intersection-2) circle (2pt) node {2};
\end{tikzpicture}
```

Coordinate systems: intersections



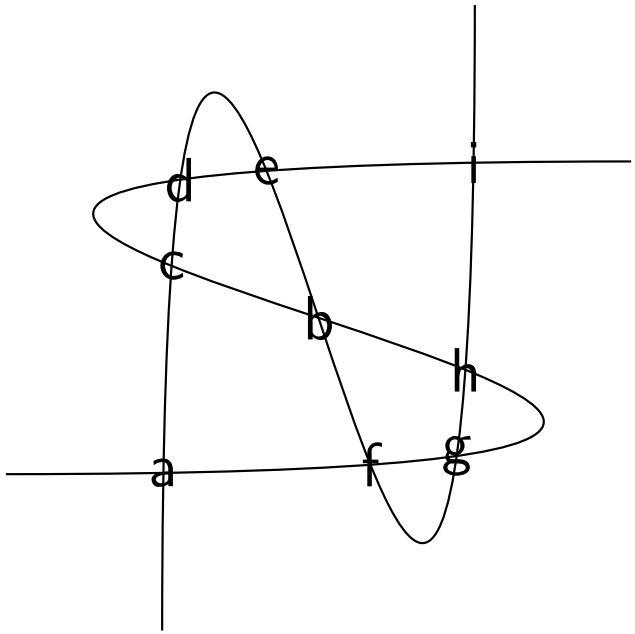

```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw[name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw[name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);
  \fill[name intersections={of=curve 1 and curve 2, name=i, total=\nInters}] %
       [red, opacity=0.5, every node/.style={above left, black, opacity=1}]
  \foreach \s in {1,...,\nInters}%
       {(i-\s) circle (2pt) node {\footnotesize\s}};
\end{tikzpicture}
```

Coordinate systems: intersections



```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw[name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw[name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);
  \fill[name intersections={of=curve 1 and curve 2, by={a,b}}] %
        (a) circle (2pt) (b) circle (2pt);
\end{tikzpicture}
```

Coordinate systems: intersections




```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw[name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw[name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);
  \fill[name intersections={of=curve 1 and curve 2, %
        by={[label=center:a],[label=center:...],[label=center:i]}}];
\end{tikzpicture}
```