1 3D Tools

Ti*k*Z Library 3dtools

```
\usetikzlibrary{3dtools} % LaTeX and plain TeX
\usetikzlibrary[3dtools] % ConTeXt
```

This library provides additional tools to create 3d-like pictures. It is a collection of reasonably working tools, which however is not streamlined, and may be subject to substantial changes if the library ever happens to get further developed or published. This library loads the `3d`, `decorations` and `fpu` libraries.

Ti*k*Z has the `3d` and `perspective` libraries which deal with the projections of three-dimensional drawings. In addition there exist excellent packages like tikz-3dplot. The purpose of this library is to provide some means to manipulate the coordinates. It supports linear combinations of vectors, vector products and scalar products. It also supports "physical" coordinates and means to transform coordinates from one orthonormal basis to another orthonormal basis.

Note: Hopefully this library is only temporary and its contents will be absorbed in slightly extended versions of the `3d` and `calc` libraries.[1] The cleanest way will be to record a screen depth when "saving" a coordinate with Ti*k*Z. Some limited support of such a functionality is provided in this library, see section 1.4. However, a full-fledged realization would require changes at the level of `tikz.code.tex` and can only be done consistently if the maintainer(s) of Ti*k*Z make some fundamental changes. Even if this happens, dealing with node anchors and so on, which so far are intrinsically two-dimensional (see, however, https://github.com/ZhiyuanLck/dnnplot for an impressive counterexample) will be nontrivial because one will have to specify the orientation in 3d as far as supported. Note also that it is quite conceivable that the viewers in the future will be able to achieve 3d ordering, so, in a way, recording the screen depth (see the `screendepth` function below) will become almost mandatory at a given point.

Filing a bug report or placing a feature request. This library is currently hosted under https://github.com/marmotghost/tikz-3dtools.

Why is this library not on CTAN? First of all, ideally this library is only temporary. Secondly, it is also under construction. There are many things that users may want to be able to use, such as spherical coordinates, and which are not supported at all. Adding support for such features my require syntax changes, thus making currently working code invalid. If this is the case, it would not be helpful if the library had already been made "official".

Code examples. This manual contains a number of code examples. They do not consist of complete LaTeX documents but only excerpts. Almost all of them require the `3dtools` library but they may require additional libraries that do not get loaded by the `3dtools` library. All libraries that are required to compile a code example are indicated. Each code example can be embedded in a minimal example, which can have the form

```
\documentclass[tikz]{standalone}        also loads TikZ
\usetikzlibrary{3dtools,...}            libraries as indicated
\begin{document}
⟨code⟩                                  code from the example
\end{document}
```

It should be also possible to embed the code in generic LaTeX documents as long as in the preamble Ti*k*Z gets loaded (e.g. with `\usepackage{tikz}`) and the relevant libraries get also loaded. Note that although above `ConTeXt` gets mentioned this library has not been tested with `ConTeXt`. The only reason why `ConTeXt` gets mentioned is that this gets automatically added by `pgfmanual.code.tex`, which gets loaded to prepare this document.

1.1 Coordinate computations

The `3dtools` library has some options and styles for coordinate computations and manipulations.

[1] Note that an earlier version of the library had been uploaded to CTAN by the maintainer of Ti*k*Z at that time. The maintainer did not notify the author of the library, let alone ask for permission. Since the library was in an even worse shape at that time, it got removed by the author. For further information on this incident please consider contacting CTAN (rather than simply believing in chat messages, which are often not too accurate).

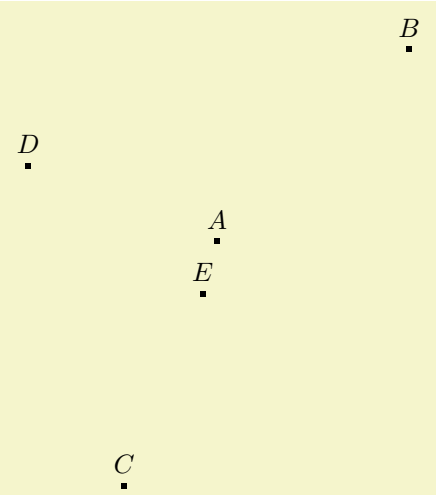
`/tikz/3d parse` (no value)

Parses an expression and inserts the result in form of a coordinate.

`/tikz/3d coordinate` (no value)

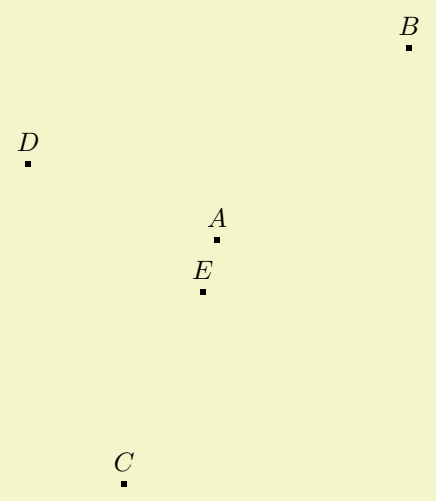
Allow one to define a 3d coordinate from other coordinates.

Both keys support both symbolic and explicit coordinates.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}
 \path (1,2,3) coordinate (A)
  (2,3,-1) coordinate (B)
  (-1,-2,1) coordinate (C)
  [3d parse={0.25*(1,2,3)x(B)}]
  coordinate(D)
  [3d parse={0.25*(C)x(B)}]
  coordinate(E);
 \path foreach \X in {A,...,E}
 {(\X) node[fill,inner sep=1pt,
 label=above:$\X$]{}};
\end{tikzpicture}
```

Notice that, as of now, only the syntax `\path (1,2,3) coordinate (A);` works, yet the syntax `\coordinate (A) at (1,2,3);` does *not* work, but leads to error messages.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}
 \path (1,2,3) coordinate (A)
  (2,3,-1) coordinate (B)
  (-1,-2,1) coordinate (C)
  [3d coordinate={(D)=0.25*(1,2,3)x(B)},
  3d coordinate={(E)=0.25*(C)x(B)},
  3d coordinate={(F)=(A)-(B)},];
 \path foreach \X in {A,...,E}
 {(\X) node[fill,inner sep=1pt,
 label=above:$\X$]{}};
\end{tikzpicture}
```

The actual parsing are done by the function `\pgfmathtdparse` that allows one to parse 3d expressions. The supported vector operations are `+` (addition +), `-` (subtraction −), `*` (multiplication of the vector by a scalar), `x` (vector product ×) and `o` (scalar product). There is limited support for ordinary PGF parsings, which are to be wrapped into `[...]`. Admittedly, this function is currently very poor. Enhancing it will be a top priority if this is ever to become a CTAN library/package.

`\pgfmathtdparse{`⟨*x*⟩`}`

Parses 3d expressions.

`TDx("`*vector*`")`

Yields the x-component of a 3d expression.

`TDy("`*vector*`")`

Yields the y-component of a 3d expression.

`TDz("vector")`

Yields the z-component of a 3d expression.

`TDunit("vector")`

Yields the result of the expression normalized to unity. If the length of the expression is too close to zero, a warning is issued and the unnormalized vector gets returned.

`screendepth("vector")`

Yields distance a coordinate is above (positive) or below (negative) the screen. The values are only really meaningful if the user has installed some reasonable view. The larger the screen depth of a point is, the closer is the point to the observer. The function reconstructs the 3-bein[2] from lengths like `\pgf@xy` and so on, so the function is independent of the tool that is employed to install a view (cf. section 1.2). The screen depth is crucial to decide whether a given object is in front of other objects, or behind.

`tddistance("point 1","point 2")`

Yields the distance between *point 1* and *point 2*. You have to keep the parentheses, e.g. `tddistance("(P)","(Q)")` works if you defined the points `(P)` and `(Q)`, but not `tddistance("P","Q")`;

`nscreenx`

Yields the x-component of the normal on the screen.

`nscreeny`

Yields the y-component of the normal on the screen.

`nscreenz`

Yields the z-component of the normal on the screen.

`x2d`

Yields the x-component of a symbolic coordinate on the screen.

`y2d`

Yields the y-component of a symbolic coordinate on the screen.

In order to pretty-print the result one may want to use `\pgfmathprintvector`, and use the math function `TD` for parsing.

`\pgfmathprintvector{`$\langle x \rangle$`}`

Pretty-prints vectors.

$0.2\,\vec{A} - 0.3\,\vec{B} + 0.6\,\vec{C} = (-1, -1.7, 1.5)$

```
\usetikzlibrary {3dtools}
\pgfmathparse{TD("0.2*(A)
-0.3*(B)+0.6*(C)")}%
$0.2\,\vec A-0.3\,\vec B+0.6\,\vec C
=(\pgfmathprintvector\pgfmathresult)$
```

The alert reader may wonder why this works, i.e. how would Ti*k*Z "know" what the coordinates A, B and C are. It works because the coordinates in Ti*k*Z are global, so they get remembered from the above example.

Warning. The expressions that are used in the coordinates will only be evaluated when they are retrieved. So, if you use, say, random numbers, you will get each time a *different* result. This is because this library is working with `\tikz@dcl@coord@`$\langle coord \rangle$, where $\langle coord \rangle$ is the name of the coordinate, say `A`. This is the string with which the coordinate was generated, e.g. `(1,2,3)`. However, the coordinate will always be at the same location. So you may want to avoid using functions that change their values when declaring coordinates that get used later in coordinate calculations.

[2] A 3-bein or dreibein is a German word that stands for a local frame, its literal translation is something like three-leg. Like the term eigenvector this is a foreign word that, to the best of my knowledge, has no commonly used English counterpart.

$\vec{R}=(0.63,0.11,1.03)$

$\vec{R}=(0.87,0.62,0.42)$

```
\usetikzlibrary {3dtools}
\begin{tikzpicture}
 \path[overlay] (0.1+rnd,0.1+rnd,0.1+rnd)
 coordinate (R);
 \def\ppv{\pgfmathprintvector\pgfmathresult}
 \draw[red] (R) circle[radius=0.1];
 \node at (0,1)
  {\pgfmathparse{TD("(R)")}%
  $\vec R=(\ppv)$};
 \draw[blue] (R) circle[radius=0.2];
 \node at (0,0)
  {\pgfmathparse{TD("(R)")}%
  $\vec R=(\ppv)$};
\end{tikzpicture}
```

The main usage of `\pgfmathprintvector` is to strip off unnecessary zeros, which emerge since the internal computations are largely done with the `fpu` library.

$(1,0,0)^T\times(0,1,0)^T=(0,0,1)^T$

```
\usetikzlibrary {3dtools}
\pgfmathparse{TD("(1,0,0)x(0,1,0)")}%
$(1,0,0)^T\times(0,1,0)^T=
(\pgfmathprintvector\pgfmathresult)^T$
```

$\vec{A}\cdot\vec{B}=5$

```
\usetikzlibrary {3dtools}
\pgfmathparse{TD("(A)o(B)")}%
$\vec A\cdot \vec B=
\pgfmathprintnumber\pgfmathresult$
```

Notice that, as of now, the only purpose of brackets `(...)` is to delimit vectors. Further, the addition `+` and subtraction `-` have a *higher* precedence than vector products `x` and scalar products `o`. That is, `(A)+(B)o(C)` gets interpreted as $(\vec{A}+\vec{B})\cdot\vec{C}$, and `(A)+(B)x(C)` as $(\vec{A}+\vec{B})\times\vec{C}$.

$(\vec{A}+\vec{B})\cdot\vec{C}=-11$

```
\usetikzlibrary {3dtools}
\pgfmathparse{TD("(A)+(B)o(C)")}%
$(\vec A+\vec B)\cdot\vec C=
\pgfmathprintnumber\pgfmathresult$
```

$(\vec{A}+\vec{B})\times\vec{C}=(9,-5,-1)$

```
\usetikzlibrary {3dtools}
\pgfmathparse{TD("(A)+(B)x(C)")}%
$(\vec A+\vec B)\times\vec C=
(\pgfmathprintvector\pgfmathresult)$
```

You can use square brackets `[...]` to separately parse subexpressions with the standard PGF parser.

$\sin(45)\,\vec{A}+\cos(30)\,\vec{B}=(2.44,4.01,1.26)^T$

```
\usetikzlibrary {3dtools}
\pgfmathparse{TD("{[sin(45)]*(A)+[cos(30)]*(B)}")}%
$\sin(45)\,\vec A+\cos(30)\,\vec B=
(\pgfmathprintvector\pgfmathresult)^T$
```

Moreover, any expression can only have either one `o` or one `x`, or none of these. Expressions with more of these can be accidentally right.

`axisangles("`*vector*`")`

Yields the the rotation angles that transforms the vector in the z-axis. Since an axis has a residual rotation symmetry, namely the rotation around this axis, only two angles are required, and thus returned. In the conventions of section 1.2, these are the angles ϕ and ψ. It corresponds to the macro `\tdplotgetpolarcoords{`$\langle x\rangle$`}{`$\langle y\rangle$`}{`$\langle z\rangle$`}` from the tikz-3dplot package.

$\sphericalangle(\vec{A})=-116.57,-36.7$

```
\usetikzlibrary {3dtools}
\pgfmathparse{axisangles("(A)")}%
$\sphericalangle(\vec A)=
\pgfmathprintvector\pgfmathresult$
```

Computations in 3d sometimes involve projections of a point on a line or a plane. In 2d, projections of a point on a line can be conveniently done with the `calc` library, but this does in general not yield

the correct projection in 3d. In order to make these projections more user friendly, `3dtools` offers the possibility to define extended objects such as lines or planes. This is the same concept as what is offered by plain Ti*k*Z, where the user can name coordinates and nodes, and, if the `intersections` library is loaded, also paths.

`/tikz/3d/plane through`=*point 1* and *point 2* and *point 3* named *name* (no default)

Defines a plane of name *name* by the requirement that it goes through the three specified points. Internally the plane is stored in terms of its normal and a point it goes through.

`/tikz/3d/plane with normal`=*normal* through *point 1* named *name* (no default)

Defines a plane of name *name* by the requirement that it goes through *point 1* and has the normal *normal*. Internally the plane is stored in terms of this normal and point.

`/tikz/3d/line through`=*point 1* and *point 2* named *name* (no default)

Defines a line of name *name* by the requirement that it goes through *point 1* and *point 1*.

`/tikz/3d/line with direction`=*vector* through *point 1* named *name* (no default)

Defines a line of name *name* by the requirement that it goes through *point 1* and has direction *vector*.

`/tikz/3d/sphere with center`=*center* and radius *radius* named *name* (no default)

Defines a sphere with center *center* and radius *radius*.

`/tikz/3d/intersection of`=*object 1* with *object 2* (no default)

Computes the intersection of a named object *object 1* with another object *object 2*. Currently only intersections of lines with lines and planes with lines are supported. There is also some support for intersections of lines with spheres. The intersections get also named by some predefined names. Note that if intersections cannot be found, warnings will be issued. Due to the limited precision of TEX the decision whether or not an intersection actually exists may be wrong in close cases.

`/tikz/3d/aux keys/intersection 1` (initially `I-1`)

Predefined name for the first intersection.

`/tikz/3d/aux keys/intersection 2` (initially `I-2`)

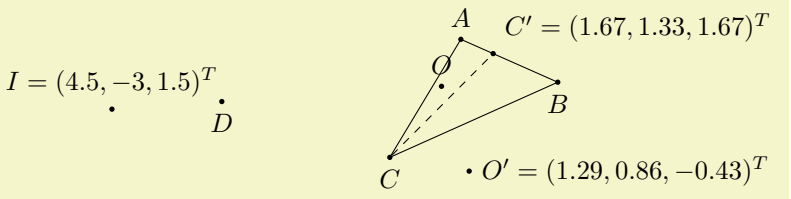
Predefined name for the second intersection.

These extended objects can be used in projections.

`/tikz/3d/project`=*point* on *named object* (no default)

Computes the projection of a point on some extended object, and inserts the projection in the path.

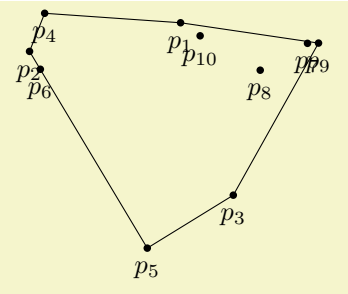
The following code example illustrates the usage. It also makes use of the `install view` key, which we describe in section 1.2.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={phi=110,psi=0,theta=60},
dot/.style={circle,inner sep=0pt,
minimum size=2pt,fill}]
 \draw[every coordinate node/.append style={dot}]
   (2,1,2) coordinate[label=above:{$A$}] (A) --
   (1,2,1) coordinate[label=below:{$B$}] (B) --
   (2,0,0) coordinate[label=below:{$C$}] (C) -- cycle
   (0,0,0) coordinate[label=above:{$O$}] (O)
   (3,-2,1) coordinate[label=below:{$D$}] (D);
 \path[3d/plane through={(A) and (B) and (C) named pABC},
3d/plane with normal={(1,1,1) through (C) named ptwo},
3d/line through={(A) and (B) named lAB},
3d/line through={(O) and (D) named lOD}];
 % project point on plane
 \path[3d/project={(O) on pABC}] coordinate (O');
 % project point on line
 \path[3d/project={(C) on lAB}] coordinate (C');
 % intersection of plane and line
 \path[3d/intersection of={lOD with pABC}] coordinate (I);
 \draw[dashed] (C) -- (C')
  coordinate[dot,label=above right:{$C'=\pgfmathparse{TD("(C')")}%
  (\pgfmathprintvector\pgfmathresult)^T$}];
 \path (O') coordinate[dot,label=right:{$O'=\pgfmathparse{TD("(O')")}%
  (\pgfmathprintvector\pgfmathresult)^T$}];
 \path (I) coordinate[dot,label=above:{$I=\pgfmathparse{TD("(I)")}%
  (\pgfmathprintvector\pgfmathresult)^T
  $}];
\end{tikzpicture}
```

`/tikz/convex hull of={`⟨*list of coordinates*⟩`}` (no default)

Style that inserts a path for the 2-dimensional convex hull of a list of named coordinates.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}
\path foreach \X in {1,...,10}
{ (rnd*360:rnd*3) coordinate (p\X)
node[circle,inner sep=1pt,fill,label={below:$p_{\X}$}]{}};
\draw[convex hull of={p1,p2,p3,p4,p5,p6,p7,p8,p9,p10}];
\end{tikzpicture}
```

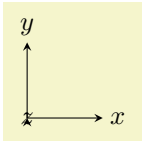
1.2 Orthonormal projections

The `3dtools` library can be used together with the `tikz-3dplot` package and/or the `perspective` library. It also has its own means to install orthonormal projections. Orthonormal projections emerge from subjecting 3-dimensional vectors to orthogonal transformations and projecting them to 2 dimensions. They are not to be confused with the perspective projections, which are more realistic and supported by the `perspective` library. Orthonormal projections may be thought of a limit of perspective projections at large distances, where large means that the distance of the observer is much larger than the dimensions of the objects that get depicted.

`/tikz/3d/install view` (no value)

Installs a 3d orthonormal projection.

The initial projection is such that x is right an y is up, as if we had no third direction.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view]
 \draw[-stealth] (0,0,0) -- (1,0,0)
  node[pos=1.2] {$x$};
 \draw[-stealth] (0,0,0) -- (0,1,0)
  node[pos=1.2] {$y$};
 \draw[-stealth] (0,0,0) -- (0,0,1)
  node[pos=1.2] {$z$};
\end{tikzpicture}
```

The 3d-like pictures emerge by rotating the view. The conventions for the parametrization of the orthogonal rotations in terms of three rotation angles ϕ, ψ and θ are

$$O(\phi,\psi,\theta) = \begin{pmatrix} c_\phi\, c_\psi & s_\phi\, c_\psi & -s_\psi \\ c_\phi\, s_\psi\, s_\theta - s_\phi\, c_\theta & s_\phi\, s_\psi\, s_\theta + c_\phi\, c_\theta & c_\psi\, s_\theta \\ c_\phi\, s_\psi\, c_\theta + s_\phi\, s_\theta & s_\phi\, s_\psi\, c_\theta - c_\phi\, s_\theta & c_\psi\, c_\theta \end{pmatrix} . \tag{1}$$

Here, $c_\phi := \cos\phi$, $s_\phi := \sin\phi$ and so on.

`/tikz/3d/phi` (initially `0`)

3d rotation angle.

`/tikz/3d/psi` (initially `0`)

3d rotation angle.

`/tikz/3d/theta` (initially `0`)

3d rotation angle.

The rotation angles can be used to define the view. The conventions are chosen in such a way that they resemble those of the tikz-3dplot package, which gets widely used. This matrix can be written as

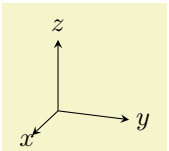
$$O(\phi,\psi,\theta) = R_x(\theta)\cdot R_y(\psi)\cdot R_z(\phi)\ ,$$

where

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{pmatrix}\ , \qquad R_y(\psi) = \begin{pmatrix} \cos(\psi) & 0 & -\sin(\psi) \\ 0 & 1 & 0 \\ \sin(\psi) & 0 & \cos(\psi) \end{pmatrix}\ ,$$

$$R_z(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix}$$

are rotations about the x, y and z axis, respectively. For $\psi = 0$, $O(\phi = \phi_d, \psi = 0, \theta = \theta_d) = R^d(\theta_d, \phi_d)$ from the tikz-3dplot package. Note, however, that there seems to be an inconsistency in equation (2.1) of that package.[3]




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={phi=110,psi=0,theta=70}]
 \draw[-stealth] (0,0,0) -- (1,0,0)
  node[pos=1.2] {$x$};
 \draw[-stealth] (0,0,0) -- (0,1,0)
  node[pos=1.2] {$y$};
 \draw[-stealth] (0,0,0) -- (0,0,1)
  node[pos=1.2] {$z$};
\end{tikzpicture}
```

`/tikz/3d/define orthonormal dreibein=`⟨*options*⟩ (no default, initially `empty`)

Defines a local dreibein from three input points. The initial choice of these points are `A=(A)`, `B=(B)`, `C=(C)`, and the initial choice for the basis vectors is `ex=(ex)`, `ey=(ey)`, `ez=(ez)`. In terms of these points, (ex) is parallel to $(B)-(A)$, (ey) is in the plane that contains (A), (B) and (C), and is orthogonal to (ex), and (ez) is orthogonal to (ex) and (ey). All the basis vectors are normalized to length 1.

[3] I do not know how to contact the author.

/tikz/3d/screen coords (no value)

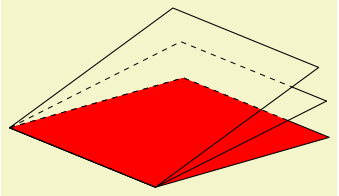
Defines locally screen coordinate system, i.e. this style brings you back to the original coordinate system in `tikzpictures`. It is taken from `tikz-3dplot`, where this style is called `tdplot_screen_coords`.

1.3 Visibility considerations

When drawing objects in 3d, some parts are "visible", i.e. in the foreground, while other parts are "hidden", i.e. in the background. In what follows, we discuss some basic means that allow one to determine whether some stretch is visible or hidden. Very often this amounts to find critical values of some parameter(s) (such as angles) at which some curve changes from visible to hidden, or vice versa. One of the major players in this game is the normal on the screen, which is given by the last row of the orthogonal matrix O of (1). For $\psi = 0$, this normal is also the normal in the so-called "main" coordinates of `tikz-3dplot` with the identifications $\theta =$ `\tdplotmaintheta` and $\phi =$ `\tdplotmainphi`.

/tikz/3d/draw ordered paths=⟨*list*⟩ (no default, initially `empty`)

This draws a list of named paths that have been created in with the `save named path` in the order they appear in the list ⟨*list*⟩. This is achieved by a number of clips and inverted clips. Notice that, at this point, layers are not supported. This is because clips have to be applied in the some layer as the path that is to be clipped against. One can define styles that carry the same names as the named paths in the directory `3d/ordered paths/`. If such a style exists, it will be applied.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={phi=50,theta=70},
  declare function={a=3;alpha=20;},
  line join=round]
 \path
   (-a/2,0,0) coordinate (P_1)
   (a/2,0,0) coordinate (P_2)
   (-a/2,a,0) coordinate (A_1)
   (a/2,a,0) coordinate (A_2)
   (-a/2,{a*cos(alpha/2)},{a*sin(alpha/2)}) coordinate (B_1)
   (a/2,{a*cos(alpha/2)},{a*sin(alpha/2)}) coordinate (B_2)
   (-a/2,{a*cos(alpha)},{a*sin(alpha)}) coordinate (C_1)
   (a/2,{a*cos(alpha)},{a*sin(alpha)}) coordinate (C_2);
 \foreach \X in {A,B,C}
 {\path[save named path=p\X] (P_1) -- (P_2) --
 (\X_2) -- (\X_1) -- cycle;}
 \pgfmathsetmacro{\nA}{TDunit("(A_1)-(P_1)x(P_2)-(P_1)")}
 \pgfmathsetmacro{\nC}{TDunit("(C_1)-(P_1)x(P_1)-(P_2)")}
 \pgfmathtruncatemacro{\itest}{screendepth(\nA)>screendepth(\nC)?1:0}
 \tikzset{3d/ordered paths/pA/.style={fill=red}}
 \ifnum\itest=1
  \tikzset{3d/draw ordered paths={pC,pB,pA}}
 \else
  \tikzset{3d/draw ordered paths={pA,pB,pC}}
 \fi
\end{tikzpicture}
```

1.3.1 Example 1: latitude circle on sphere

Here is an example. Suppose we wish to draw a latitude circle on a sphere for $\psi = 0$. Since the system has a rotational symmetry around the z axis, we can set $\phi = 0$ as well. The normal on the screen then becomes

$$\vec{n}_{\text{screen}} = \big(0, -\sin(\theta), \cos(\theta)\big)^T .$$

The latitude circle can be parametrized as

$$\vec{\gamma}_{\text{lat}}(\beta) = \big(r\,\cos(\beta), r\,\sin(\beta), h\big)^T ,$$

where $r = R\cos(\lambda)$ and $h = R\sin(\lambda)$ with R being the radius of the sphere and λ the latitude angle of the latitude circle. The critical angles β_{crit} at which the curve transitions from hidden to visible are then given by the solutions of

$$\vec{n}_{\text{screen}} \cdot \vec{\gamma}_{\text{lat}}(\beta_{\text{crit}}) = -r\,\sin(\beta)\,\sin(\theta) + h\,\cos(\theta) \stackrel{!}{=} 0 .$$

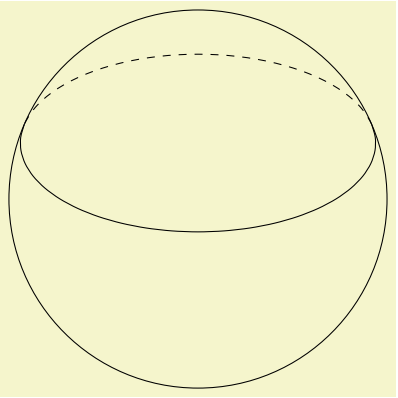
Since $-1 \leq \sin(\beta) \leq 1$, this equation only has solutions if

$$\left|\frac{h}{r}\cot(\theta)\right| \leq 1\,. \tag{2}$$

The solutions are then given by

$$\beta^{(1)}_{\text{crit}} = \arcsin\left(\frac{h}{r}\cot(\theta)\right) \quad \text{and} \quad \beta^{(2)}_{\text{crit}} = 180 - \beta^{(1)}_{\text{crit}}\,,$$

and $\beta^{(1)}_{\text{crit}} = \beta^{(2)}_{\text{crit}}$ if the inequality (2) is saturated. The following code example illustrates this. Notice that it could be made more concise with the `3d` library.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[declare function={%
R=2.5;lambda=20;theta=60;
r=R*cos(lambda);h=R*sin(lambda);
betacrit=asin(h/r*cot(theta));}]
 \draw (0,0) circle[radius=R];
 \begin{scope}[smooth,
 3d/install view={phi=0,psi=0,theta=theta}]
  \draw[dashed]
   plot[variable=\t,
   domain=betacrit:180-betacrit]
   ({r*cos(\t)},{r*sin(\t)},h);
  \draw
   plot[variable=\t,
   domain=betacrit:-180-betacrit]
   ({r*cos(\t)},{r*sin(\t)},h);
 \end{scope}
\end{tikzpicture}
```

1.3.2 Example 2: arbitrary circle on sphere

A somewhat more advanced question is to draw an arbitrary circle on a sphere. That is, we are given a sphere around a center C or radius R and a point P inside the sphere. If C and P coincide, we need to define a normal n, otherwise $n = P - C$. How can one draw a circle on the sphere, distinguishing between visible and hidden stretches?

First test: $|P - C| < R$. The circle only makes sense if $|P - C| < R$. The radius of the circle is $r = \sqrt{R^2 - |P - C|^2}$. Call the normal on the screen n_{screen}.

Case $n \parallel n_{\text{screen}}$. If n and n_{screen} are linearly dependent, the circle is completely hidden if the screen depth of P is smaller than the screen depth of C, $\text{sd}\,P < \text{sd}$, and visible otherwise. We can choose a coordinate system in which

$$e_z = \frac{n}{|n|}\,, \quad e_x \text{ arbitrary but } \perp e_z \quad \text{and} \quad e_y = e_z \times e_x$$

to draw the circle.

Case $n \nparallel n_{\text{screen}}$. From now on we assume that n and n_{screen} are not linearly dependent. This means that they span a plane, and we can choose n_{screen} to point in the x-direction of this plane (see figure 1).

The normal can be decomposed in a part parallel to n_{screen}, $n_\parallel$, and a perpendicular part, $n_\perp$. The projection of the circle on this plane is a line, and has slope $\alpha = \arctan(-n_\parallel/n_\perp)$. If $r\cos\alpha > |\text{sd}\,P - \text{sd}\,C|$, the circle has both hidden and visible stretches, from between $-r\cos\alpha$ and $\text{sd}\,P - \text{sd}\,C$ it is hidden and between $\text{sd}\,P - \text{sd}\,C$ and $r\cos\alpha$ it is solid.

We are going to draw the circle in a coordinate system defined by

$$e_z = \frac{n}{|n|}\,, \quad e_y = e_z \times n_{\text{screen}} \quad \text{and} \quad e_x = e_y \times e_z\,.$$

The circle is then in the x-y plane, and the positive x-direction is the direction of increasing screen depth (or visibility). If $r\cos\alpha > |\text{sd}\,P|$, then the stretch between $\beta := \left|\arccos\left(\frac{\text{sd}\,C - \text{sd}\,P}{r\cos\alpha}\right)\right|$ and $-\beta$ is visible, between β and $360° - \beta$ it is hidden.

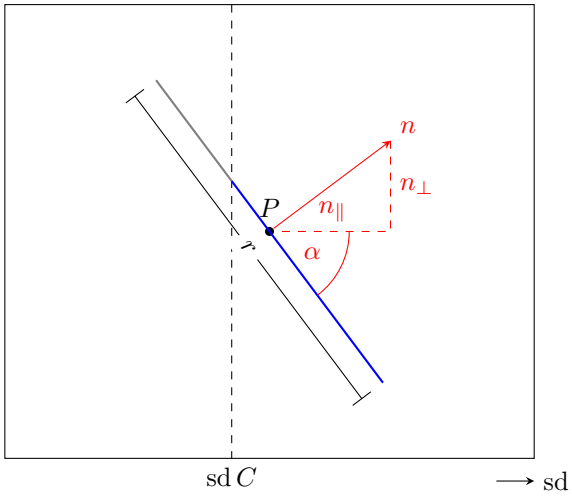


Figure 1: $n \not\parallel n_{\text{screen}}$.

1.3.3 Example 3: outlines of shapes

Another type of visibility considerations concerns the contours of shapes. To be specific, consider a surface parametrized by

$$\vec{f}(u,v) = \begin{pmatrix} u \\ R(u)\,\cos(v) \\ R(u)\,\sin(v) \end{pmatrix} \tag{3}$$

with some function $R(u)$. What is the contour of an orthonormal projection of the surface on the screen? It is the boundary between the hidden and visible parts. What distinguishes the visible from the hidden parts? It is the projection of the normal of the surface on the screen, $F(u,v) := n_S(u,v) \cdot n_{\text{screen}}$. The hidden and visible stretches are separated by the zeros of this projection. In the case of our surface (3),

$$\vec{n}_S(u,v) = \vec{\nabla}_u f(u,v) \times \vec{\nabla}_v f(u,v) = R(u) \begin{pmatrix} -R'(u) \\ \cos(v) \\ \sin(v) \end{pmatrix} , \tag{4}$$

and for $\psi = 0$

$$\vec{n}_{\text{screen}} = \begin{pmatrix} \sin(\theta)\,\sin(\phi) \\ -\sin(\theta)\,\cos(\phi) \\ \cos(\theta) \end{pmatrix} . \tag{5}$$

So we need to find the zeros of

$$\begin{aligned} F(u,v) = & -\cos(\theta)\,\sin(v) + \sin(\theta)\,\cos(\phi)\,\cos(v) + R'(u)\,\sin(\theta)\,\sin(\phi) \\ =: & \; a\,\sin(v) + b\,\cos(v) + c\,. \end{aligned} \tag{6}$$

This equation can be solved for $x := \sin(v)$ using $\cos(v) = \pm\sqrt{1-\sin^2(v)} = \pm\sqrt{1-x^2}$. It is clear that, since $\sin(v)$ is bounded (and so is $\cos(v)$), this equation does not always have a solution. In particular, if $R'(u)$ is too large, there might not be a solution. This just means that for a given u all points are hidden or visible. This happens e.g. when you look at a cone from the top or bottom. A more advanced code could deal with these cases, the one presented below (which is taken from topanswers.xyz) does not. Note also that these are local visibility conditions, a seemingly visible stretch can always be covered by some other stretch that is a finite distance away in u direction.

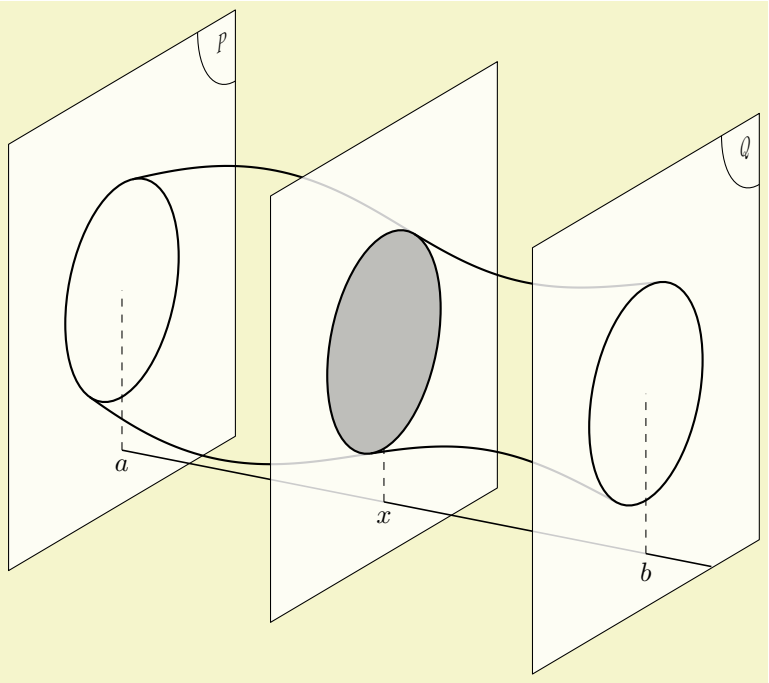
However, for "reasonable" configurations, there are two solutions of the quadratic equation, as they should, they correspond to the upper and lower contour in the code example below. They are a bit

unilluminating and given by

$$[\sin(v)]_{1,2} = -\frac{a\sqrt{a^2+b^2-c^2} \pm b\,c}{a^2+b^2}\,, \tag{7a}$$

$$[\cos(v)]_{1,2} = -\frac{b\sqrt{a^2+b^2-c^2} \mp a\,c}{a^2+b^2}\,, \tag{7b}$$

where a, b and c are implicitly defined in (6). Hence, for "reasonable" configurations, it is straightforward to plot the contour of the surface with LaTeX.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[declare function={phi=30;theta=70;},
3d/install view={phi=phi,psi=0,theta=theta},
line cap=butt,line join=round,
declare function={%
  R(\u)=1.5+sin(\u*45)/(2+\u/8);%<- input
  Rprime(\u)=(R(\u+0.01)-R(\u-0.01))/0.02;%<- numerical derivative
  a=cos(phi)*sin(theta);% see equation {eq:vcrit}
  b=-1*cos(theta);%
  c(\u)=sin(theta)*sin(phi)*Rprime(\u);%
      sv1(\u)=-((b*c(\u)+sqrt(a*a+b*b-c(\u)*c(\u))*abs(a))/(a*a+b*b));%
  sv2(\u)=(-(b*c(\u))+sqrt(a*a+b*b-c(\u)*c(\u))*abs(a))/(a*a+b*b);%
  cv1(\u)=-((a*c(\u)+sqrt(a*a+b*b-c(\u)*c(\u))*abs(b))/(a*a+b*b));%
  cv2(\u)=(-(a*c(\u))+sqrt(a*a+b*b-c(\u)*c(\u))*abs(b))/(a*a+b*b);%
},
icircle/.style={thick},
iplane/.style={fill=white,fill opacity=0.8},
iline/.style={semithick},
 extra/.code={},annot/.code={\tikzset{extra/.code={#1}}}]
 \newcommand\DrawIntersectingPlane[2][]{%
  \begin{scope}[canvas is yz plane at x=#2,transform shape,#1]
 \draw[iplane] (3,3) -- (-3,3) -- (-3,-3)-- (3,-3) -- cycle;
 \draw[icircle]  (0,0) circle[radius={R(#2)}];
 \tikzset{extra}
  \end{scope}}
 %
 \DrawIntersectingPlane[annot={%
 \path (2.9,2.9) node[below left,transform shape]{$P$};
\draw (2,3) arc[start angle=180,end angle=270,radius=1];
\draw[dashed] (0,-2.25) node[below,transform shape=false]{$a$} -- (0,0);
}]{0}
 \draw[iline] (0,0,-2.25) -- (4,0,-2.25);
 \draw[thick,smooth,variable=\u,domain=0:4]
 plot  (\u,{R(\u)*cv1(\u)},{R(\u)*sv1(\u)})
 plot  (\u,{R(\u)*cv2(\u)},{R(\u)*sv2(\u)});
 \DrawIntersectingPlane[icircle/.append style={fill=gray,fill opacity=0.5},
   annot={%
\draw[dashed] (0,-2.25) node[below,transform shape=false]{$x$} -- (0,-1.5);
}]{4}
 \draw[iline] (4,0,-2.25) -- (8,0,-2.25);
 \draw[thick,smooth,variable=\u,domain=4:8]
 plot  (\u,{R(\u)*cv1(\u)},{R(\u)*sv1(\u)})
 plot  (\u,{R(\u)*cv2(\u)},{R(\u)*sv2(\u)});
 %
 \DrawIntersectingPlane[annot={%
 \path (2.9,2.9) node[below left,transform shape]{$Q$};
\draw (2,3) arc[start angle=180,end angle=270,radius=1];
\draw[dashed] (0,-2.25) node[below,transform shape=false]{$b$} -- (0,0);
}]{8}
 \draw[iline] (8,0,-2.25) -- (9,0,-2.25) ;
\end{tikzpicture}
```

There are some special cases that may be of particular interest: $R'(u) = 0$ for a cylinder and $R'(u) = -r/h$ for a cone of height h and base radius r.

1.4 "Physical" coordinates

It is quite conceivable that users may want to use different coordinate systems for defining different coordinates. The following functions aim at supporting this. It should be said, though, that these functions are even more "experimental" than what has been discussed thus far. The main point is that, apart from the "physical" x and y coordinates, i.e. the coordinates of an point on the screen, one can also keep track of the physical z coordinate, or screen depth, which indicates how far a given point is above ($z > 0$) or below ($z < 0$) the screen. To accomplish this, one needs to switch on the recording of physical components.

`/tikz/3d/record physical components` (no value)

Switches on recording of physical components of coordinates.

The physical components can then be retrieved with various functions.

`TDphysx("vector")`

Yields the physical x-component of a 3d named coordinate.

`TDphysy("vector")`

Yields the physical y-component of a 3d named coordinate.

`TDphysz("vector")`

Yields the physical z-component of a 3d named coordinate.

`TDphys("vector")`

Yields an array containing physical x, y and z-components of a 3d named coordinate.

Sometimes one may want to retrieve the components of a physical coordinate in a local frame.

`TDlocalx("vector")`

Yields the local x-component of a 3d named coordinate the physical coordinates of which have been recorded.

`TDlocaly("vector")`

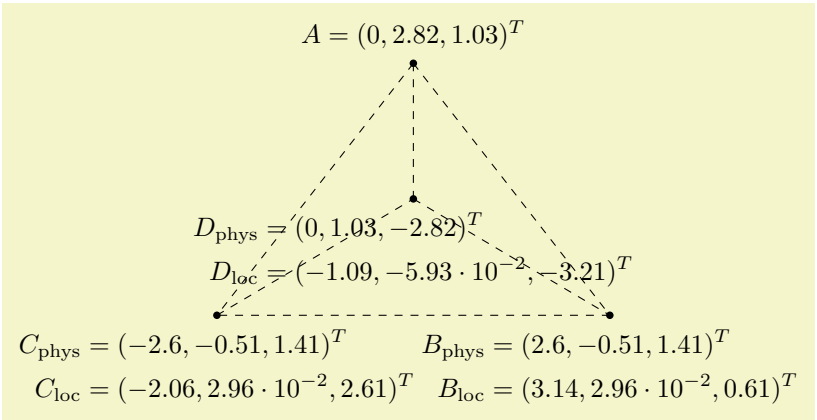
Yields the local y-component of a 3d named coordinate the physical coordinates of which have been recorded.

`TDlocalz("vector")`

Yields the local z-component of a 3d named coordinate the physical coordinates of which have been recorded.

`TDlocal("vector")`

Yields an array containing local x, y and z-components of a 3d named coordinate the physical coordinates of which have been recorded.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/record physical components,
dot/.style={circle,inner sep=1pt,fill}]
 \begin{scope}[3d/install view={phi=0,psi=0,theta=70}]
  \path (0,0,3) coordinate (A);
  \path[3d/install view={phi=30}] (3,0,0) coordinate (B);
  \path[3d/install view={phi=150}] (3,0,0) coordinate (C);
  \path[3d/install view={phi=270}] (3,0,0) coordinate (D);
 \end{scope}
 \draw[dashed] foreach \X [remember=\X as \Y (initially D)]
 in {B,C,D}
  {(A) -- (\X) (\Y) -- (\X)
 coordinate[dot,label=below:{$\begin{aligned}
\X_\mathrm{phys}&=\pgfmathparse{TDphys("\X")}%
 (\pgfmathprintvector\pgfmathresult)^T\\
\X_\mathrm{loc}&=\pgfmathparse{TDlocal("\X")}%
 (\pgfmathprintvector\pgfmathresult)^T\\
 \end{aligned}$}]}
 (A) coordinate[dot,label=above:{$A=\pgfmathparse{TDphys("A")}%
 (\pgfmathprintvector\pgfmathresult)^T
 $}];
\end{tikzpicture}
```

Once one has defined physical coordinates, one can use them to compute e.g. distances between coordinates defined in different frames.

$$\vec{A}_{\mathrm{phys}} = (0, 2.82, 1.03),$$
$$\vec{B}_{\mathrm{phys}} = (2.6, -0.51, 1.41),$$
$$d_{\mathrm{phys}}(\vec{A}, \vec{B}) = 4.24$$

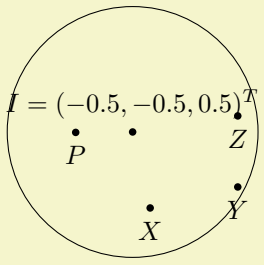
```
\usetikzlibrary {3dtools}
$\pgfmathsetmacro{\vecA}{TDphys("A")}%
\pgfmathsetmacro{\vecB}{TDphys("B")}%
\pgfmathsetmacro{\physdist}{%
sqrt(TD("(\vecA)-(\vecB)o(\vecA)-(\vecB)"))}%
\begin{array}{l}
\vec A_\mathrm{phys}=(\pgfmathprintvector\vecA),\\
\vec B_\mathrm{phys}=(\pgfmathprintvector\vecB),\\
d_\mathrm{phys}(\vec A,\vec B)=%
\pgfmathprintnumber\physdist
\end{array}$
```

These are the things that work. However, there are, unfortunately, more than enough things that do not work. They include shifted frames, other coordinate systems such as spherical ones, all coordinates that are defined in `canvas is xy plane at z=0` or relatives, etc.

1.5 Predefined path constructions and pics

`/tikz/3d/circumsphere center=`⟨*options*⟩ (no default, initially `empty`)

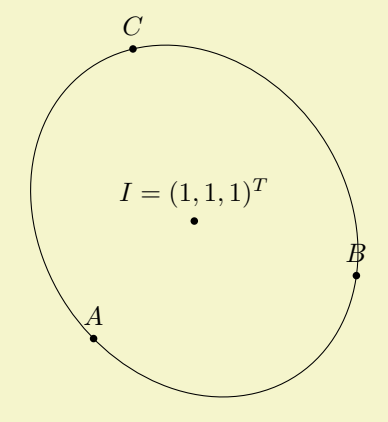
Computes the center of a sphere for a given set of four non-coplanar points. The initial choice of these points are `A=(A)`, `B=(B)`, `C=(C)`, `D=(D)`. The underlying maths can be found at https://topanswers.xyz/tex?q=1233#a1467.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[dot/.style={circle,inner sep=1pt,fill}]
 \begin{scope}[3d/install view={phi=100,psi=0,theta=70}]
  \path (1,0,0) coordinate (X)
   (0,1,0) coordinate (Y)
   (0,1,1) coordinate (Z)
   (1,-1,1) coordinate (P);
  \path[3d/circumsphere center={A={(X)},B={(Y)},C={(Z)},D={(P)}}]
  coordinate (I);
 \end{scope}
  \pgfmathsetmacro{\csr}{sqrt(TD("(X)-(I)o(X)-(I)"))}
  \draw (I) circle[radius=\csr];
  \path foreach \X in {X,Y,Z,P}
 {(\X) coordinate[dot,label=below:{$\X$}]}
 (I) (I) coordinate[dot,label=above:{$I=\pgfmathparse{TD("(I)")}%
  (\pgfmathprintvector\pgfmathresult)^T$}];
\end{tikzpicture}
```

/tikz/3d/circumcircle center=⟨*options*⟩ (no default, initially `empty`)

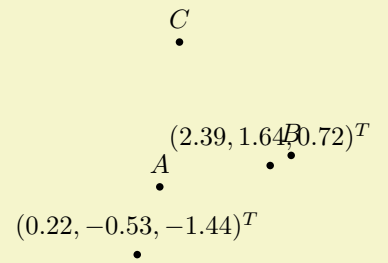
Computes the center of a circle for a given set of three non-collinear points. The initial choice of these points are `A=(A)`, `B=(B)`, `C=(C)`.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[declare function={a=3;},
   dot/.style={circle,inner sep=1pt,fill},
   3d/install view={phi=100,psi=0,theta=70}]
  \path (a,0,0) coordinate (A)
   (0,a,0) coordinate (B)
   (0,0,a) coordinate (C);
  \path[3d/circumcircle center]
  coordinate (I);
  \pgfmathsetmacro{\myr}{%
     sqrt(TD("(A)-(I)o(A)-(I)"))}
  \tikzset{3d/define orthonormal dreibein}
  \begin{scope}[x={(ex)},y={(ey)}]
   \draw (I) circle[radius=\myr];
  \end{scope}
  \path foreach \X in {A,B,C}
  {(\X) coordinate[dot,label=above:{$\X$}]}
  (I) (I) coordinate[dot,label=above:{$I=%
  \pgfmathparse{TD("(I)")}%
   (\pgfmathprintvector\pgfmathresult)^T$}];
\end{tikzpicture}
```

/tikz/3d/intersection of three spheres=⟨*options*⟩ (no default, initially `empty`)

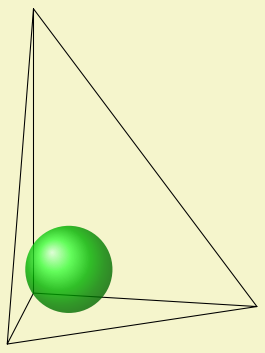
Computes the intersection intersections of three spheres around `A=(A)`, `B=(B)`, and `C=(C)`. The radii are stored in `rA`, `rB` and `rC`, respectively. The names of the intersection solutions are stored in the keys `i1` and `i2`, the initial values of which are `i1` and `i2`.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[declare function={a=1.5;},
   dot/.style={circle,inner sep=1pt,fill},
   3d/install view={phi=100,psi=0,theta=70}]
  \path (a,0,0) coordinate (A)
   (0,a,0) coordinate (B)
   (0,0,a) coordinate (C);
  \path[3d/intersection of three spheres=
  {rA=2,rB=2.5,rC=3}];
  \path (i1)
    coordinate[dot,label=above:{$%
  \pgfmathparse{TD("(i1)")}%
   (\pgfmathprintvector\pgfmathresult)^T$}];
  \path (i2)
    coordinate[dot,label=above:{$%
  \pgfmathparse{TD("(i2)")}%
   (\pgfmathprintvector\pgfmathresult)^T$}];
  \path foreach \X in {A,B,C}
  {(\X) coordinate[dot,label=above:{$\X$}]};
\end{tikzpicture}
```

/tikz/3d/insphere center=⟨*options*⟩ (no default, initially `empty`)

Computes the center of an insphere center of a tetrahedron defined by four corners, `A`, `B`, `C`, and `D`. The initial values are `A=(A)`, `B=(B)`, `C=(C)`, and `D=(D)`.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[line join=round,line cap=round,
3d/install view={phi=100,psi=0,theta=70}]
   \path (0,0,0) coordinate (D)
(2,0,0) coordinate (A)
(0,3,0) coordinate (B)
(0,0,4) coordinate (C);
   \path[3d/insphere center] coordinate (I);
   \tikzset{3d/plane through=(A) and (B) and (C) named pABC}
   \path[3d/project={(I) on pABC}] coordinate (I');
   \pgfmathsetmacro{\myr}{tddistance("(I)","(I')")}
   \draw (A) -- (B) -- (C) -- cycle --(D) --(B) (C) -- (D);
   \shade[3d/screen coords,ball color=green, opacity=0.8]
(I) circle [radius=\myr];
\end{tikzpicture}
```

/tikz/pics/3d circle through 3 points=⟨*options*⟩ (no default, initially `empty`)

Draws a circle through 3 points in 3 dimensions. If the three coordinates are close to linearly dependent, the circle will not be drawn. *This pic will most likely be removed from the documentation and no longer be supported/developed.*

/tikz/3d circle through 3 points/A (initially `(1,0,0)`)

First coordinate. Can be either symbolic or explicit. Symbolic coordinates need to be defined via `\path (x,y,z) coordinate (name);`.

/tikz/3d circle through 3 points/B (initially `(0,1,0)`)

Second coordinate, like above.

/tikz/3d circle through 3 points/C (initially `(0,0,1)`)

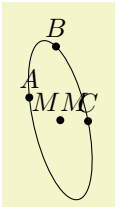
Third coordinate, like above.

/tikz/3d circle through 3 points/center name (initially `M`)

Name of the center coordinate that will be derived.

/tikz/3d circle through 3 points/auxiliary coordinate prefix (initially `tmp`)

In Ti*k*Z the coordinates are global. The code for the circle is more comprehensible if named coordinates are introduced. Their names will begin with this prefix. Changing the prefix will allow users to avoid overwriting existing coordinates.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={%
    phi=30,psi=0,theta=70}]
 \foreach \X in {A,B,C}
 {\pgfmathsetmacro{\myx}{3*(rnd-1/2)}
 \pgfmathsetmacro{\myy}{3*(rnd-1/2)}
 \pgfmathsetmacro{\myz}{3*(rnd-1/2)}
 \path (\myx,\myy,\myz) coordinate (\X);}
 \path pic{3d circle through 3 points={%
 A={(A)},B={(B)},C={(C)},center name=MM}};
  \foreach \X in {A,B,C,MM}
  {\fill (\X) circle[radius=1.5pt]
  node[above]{$\X$};}
\end{tikzpicture}
```

/tikz/pics/3d/circle on sphere=⟨*options*⟩ (no default, initially `empty`)

Draws a circle on as sphere. It distinguishes between foreground and background. Notice that this pic defines its own dreibein, so it will define coordinates that draw their names from `/tikz/3d/aux keys/ex`, `/tikz/3d/aux keys/ey` and `/tikz/3d/aux keys/ez`. It will also define a

coordinate (nscreen) which contains the normal to the screen. The foreground path will use the style /tikz/3d/visible, and the hidden path will be drawn with the style /tikz/3d/hidden. For an alternative approach with more functionality see the tikz-3dplot-circleofsphere package.

/tikz/3d/circle on sphere/C (initially (0,0,0))

Coordinate of the center of the sphere.

/tikz/3d/circle on sphere/P (initially (0,0,0))

Coordinate of the center of the circle.

/tikz/3d/circle on sphere/n (initially (0,0,1))

Normal. This is only needed if the center of the circle coincides with the center of the sphere.

/tikz/3d/circle on sphere/auxiliary coordinate prefix (initially tmp)

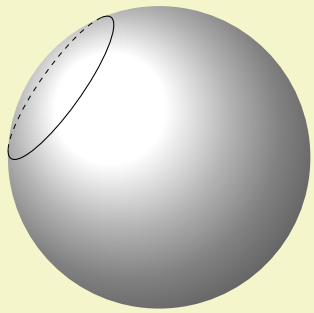
This pic installs a symbolic coordinate for the center of the circle. The auxiliary prefix can be used to avoid overwriting existing coordinates. The initial choice of the prefix will lead to a coordinate with name (tmpP).

/tikz/3d/circle on sphere/fore layer=⟨*layer*⟩ (no default, initially main)

Layer of the visible faces.

/tikz/3d/circle on sphere/back layer=⟨*layer*⟩ (no default, initially main)

Layer of the hidden faces.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={phi=110,theta=70},
declare function={R=2;}]
 \shade[ball color=white,3d/screen coords] circle[radius=R];
 \path pic{3d/circle on sphere={R=R,P={(0.5,-1.2,1)}}};
\end{tikzpicture}
```

/tikz/pics/3d incircle=⟨*options*⟩ (no default, initially empty)

Inscribes a circle in a triangle in 3 dimensions.

/tikz/3d incircle/A (initially (1,0,0))

First coordinate. Can be either symbolic or explicit.

/tikz/3d incircle/B (initially (0,1,0))

Second coordinate, like above.

/tikz/3d incircle/C (initially (0,0,1))

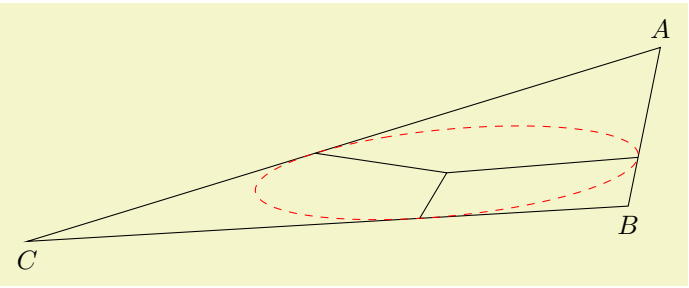
Third coordinate, like above.

/tikz/3d incircle/center name (initially I)

Name of the center coordinate that will be derived.

/tikz/3d incircle/projection prefix (initially tmpp)

In Ti*k*Z the coordinates are global. The code for the circle is more comprehensible if named coordinates are introduced. Their names will begin with this prefix. Changing the prefix allows users to avoid overwriting existing coordinates.




```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={phi=110,psi=0,theta=70}]
 \draw
   (8,5,5) coordinate[label=above:{$A$}] (A) --
   (1,2,0) coordinate[label=below:{$B$}] (B) --
   (5,-5,0) coordinate[label=below:{$C$}] (C) -- cycle;
 \path pic[red,dashed]{3d incircle={%
 A={(A)},B={(B)},C={(C)},center name=I}};
 \draw (I) -- (tmppa) (I) -- (tmppb) (I)  -- (tmppc);
\end{tikzpicture}
```

`/tikz/3d/polyhedron/draw face with corners=`⟨*list of corners*⟩ (no default, initially `empty`)

Draws a face of a polyhedron through the specified corners. It distinguishes between "visible" and "hidden" faces. The user can specify a point inside the corner. Then the outwards pointing normal of the face can either have a negative projection on the screen, in which case the face is hidden, or else it is visible.

`/tikz/3d/polyhedron/draw open face with corners=`⟨*list of corners*⟩ (no default, initially `empty`)

Same as the previous key except it does not close the surface boundary path.

`/tikz/3d/polyhedron/O=`⟨*coordinate*⟩ (no default, initially `(0,0,0)`)

Coordinate inside the polyhedron.

`/tikz/3d/polyhedron/L=`⟨*coordinate*⟩ (no default, initially `(1,1,1)`)

Coordinate of light source.

`/tikz/3d/polyhedron/shading function=`⟨*name*⟩ (no default, initially `tikztdpolyhedronshade`)

Function of the projection of the light source on the normal of the respective face that determines the shading.

`/tikz/3d/polyhedron/fore layer=`⟨*layer*⟩ (no default, initially `main`)

Layer of the visible faces.

`/tikz/3d/polyhedron/back layer=`⟨*layer*⟩ (no default, initially `main`)

Layer of the hidden faces.

`/tikz/3d/polyhedron/color=`⟨*color*⟩ (no default, initially `yellow`)

Color of the polyhedron.

`/tikz/3d/polyhedron/fore=`⟨*style*⟩ (no default, initially `draw,solid`)

Style for visible faces. One can turn of the fill by adding `fill=none`.

`/tikz/3d/polyhedron/back=`⟨*style*⟩ (no default, initially `draw,dashed,fill opacity=0`)

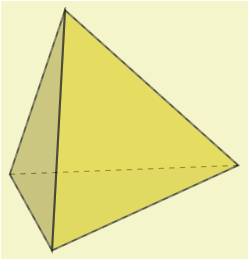
Style for visible faces.

`/tikz/3d/polyhedron/complete dashes=`⟨*style*⟩ (no default, initially `on 2pt off 2pt`)

Switches on dashes for the edges which start and end with an on phase.

`/tikz/cheating dash=`⟨*style*⟩ (no default, initially `on 2pt off 2pt`)

Switches on dashes with an on phase. Taken from Mark Wibrow's answer.



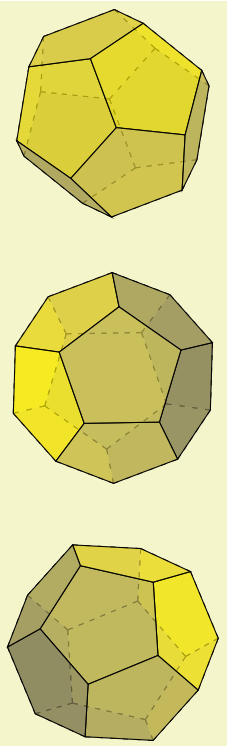
```
\usetikzlibrary {3dtools}
\pgfdeclarelayer{background}
\pgfdeclarelayer{foreground}
\pgfsetlayers{background,main,foreground}
\begin{tikzpicture}
\begin{scope}[3d/install view={phi=110,psi=10,%
theta=70},scale=2]
  \path (0,0,1) coordinate (v1)
  ({sqrt(8/9)},0,-1/3) coordinate (v2)
  ({-sqrt(2/9)},{sqrt(2/3)},-1/3) coordinate (v3)
  ({-sqrt(2/9)},{-sqrt(2/3)},-1/3) coordinate (v4);
  \tikzset{3d/polyhedron/.cd,
fore layer=foreground,back layer=background,
fore/.append style={opacity=0.4,thick},
back/.append style={3d/polyhedron/complete dashes,
opacity=0.4},
 draw face with corners={{(v1)},{(v2)},{(v3)}},
 draw face with corners={{(v1)},{(v2)},{(v4)}},
 draw face with corners={{(v1)},{(v3)},{(v4)}},
 draw face with corners={{(v2)},{(v3)},{(v4)}}
\end{scope}
\end{tikzpicture}
```

`/tikz/3d/define vertices`=⟨*list*⟩ (no default, initially `empty`)

Allows the user to define a list of vertices from a list. These vertices have names that consist of their indices in the list. One can work with the `name prefix` key to give them more meaningful names. Such lists can be generated with `Mathematica`, e.g. with `N[PolyhedronData["Dodecahedron", "VertexCoordinates"]]`.

`/tikz/3d/polyhedron/create faces from vertex list`=⟨*list*⟩ (no default, initially `empty`)

Draws a face from a list of vertices. Such lists can be generated with with `Mathematica`, e.g. with `PolyhedronData["Dodecahedron", "FaceIndices"]`. This key is very similar to `draw face with corners` except that it also adds the `name prefix`, and that it loops over the list. One should avoid spaces in the list.



```
\usetikzlibrary {3dtools}
\pgfdeclarelayer{background}\pgfdeclarelayer{foreground}
\pgfsetlayers{background,main,foreground}
\begin{tikzpicture}
 \edef\lstV{{-1.376,0.,0.2628},{1.376,0.,-0.2628},{-0.4253,-1.309,0.2628},
 {-0.4253,1.309,0.2628},{1.113,-0.8090,0.2628},{1.113,0.8090,0.2628},
 {-0.2628,-0.8090,1.113},{-0.2628,0.8090,1.113},{-0.6881,-0.5,-1.113},
 {-0.6881,0.5,-1.113},{0.6881,-0.5,1.113},{0.6881,0.5,1.113},
 {0.8506,0.,-1.113},{-1.113,-0.8090,-0.2628},{-1.113,0.8090,-0.2628},
 {-0.8506,0.,1.113},{0.2628,-0.8090,-1.113},{0.2628,0.8090,-1.113},
 {0.4253,-1.309,-0.2628},{0.4253,1.309,-0.2628}}
 \edef\lstFaces{{15,10,9,14,1},{2,6,12,11,5},{5,11,7,3,19},{11,12,8,16,7},
 {12,6,20,4,8},{6,2,13,18,20},{2,5,19,17,13},{4,20,18,10,15},{18,13,17,9,10},
 {17,19,3,14,9},{3,7,16,1,14},{16,8,4,15,1}}
 \tikzset{3d/polyhedron/.cd,fore layer=foreground,back layer=background,
fore/.append style={fill opacity=0.7},
back/.append style={3d/polyhedron/complete dashes,fill opacity=0.7}}
\begin{scope}[3d/install view={phi=120,psi=20,%
theta=70}]
 \tikzset{name prefix=Va,%<- used for all vertices
 3d/define vertices/.expanded={\lstV},
3d/polyhedron/create faces from vertex list/.expanded={\lstFaces}}
\end{scope}
\begin{scope}[yshift=-3.5cm,
3d/install view={phi=210,psi=30,theta=60}]
 \tikzset{name prefix=Vb,%<- used for all vertices
 3d/define vertices/.expanded={\lstV},
 3d/polyhedron/create faces from vertex list/.expanded={\lstFaces}}
\end{scope}
\begin{scope}[yshift=-7cm,
3d/install view={phi=30,psi=-10,theta=75}]
 \tikzset{name prefix=Vc,%<- used for all vertices
 3d/define vertices/.expanded={\lstV},
 3d/polyhedron/create faces from vertex list/.expanded={\lstFaces}}
\end{scope}
\end{tikzpicture}
```

The library has also some experimental pics for cones, truncated cones and cylinders. They share a couple of common keys.

`/tikz/3d/r` (initially `1`)

Key for radii, e.g. of cylinders or cones.

`/tikz/3d/h` (initially `1`)

Key for heights, e.g. of cylinders or cones.

`/tikz/3d/R` (initially `2`)

Key for larger radii, e.g. of a frustum.

`/tikz/pics/3d/visible` (initially `draw,solid`)

Style for visible lines.

`/tikz/pics/3d/hidden` (initially `draw,very thin,cheating dash`)

Style for hidden lines.

`/tikz/pics/3d/cone` (initially `empty`)

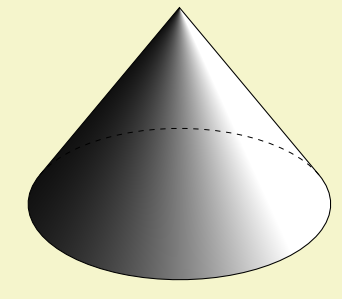
Draws a cone with the basis in the current xy plane.

`/tikz/pics/3d/shaded cone` (initially `empty`)

Draws a shaded cone with the basis in the current xy plane.

`/tikz/pics/3d/cone/shading` (initially `empty`)

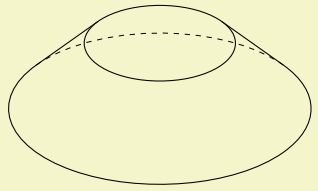
Defines and uses a functional shading that can be used for cones. The parameters are the coordinates of the tip `x` and `y`, which have to satisfy $-25 < x, y < 25$, and `alpha`, which indicates the rotation of the shading.



```
\usetikzlibrary {calc,3dtools}
\begin{tikzpicture}[3d/install view=%
{phi=110,psi=0,theta=60}]
 \pic{3d/shaded cone={r=2,h=3}};
\end{tikzpicture}
```

/tikz/pics/3d/frustum (initially empty)

Draws a frustum with the basis in the current xy plane. Notice that if both radii are equal this is just a cylinder.



```
\usetikzlibrary {calc,3dtools}
\begin{tikzpicture}[3d/install view=%
{phi=110,psi=0,theta=60}]
 \pic{3d/frustum};
\end{tikzpicture}
```

/tikz/pics/ycylinder (initially empty)

A cylinder in the y-direction. This pic requires the `calc` library. As of now it does only work for `psi=0`. This key is obsolete because of the `cylinder` pic which allows the user to dial an arbitrary axis.

/tikz/pics/cylinder (initially empty)

A cylinder with an arbitrary axis. This pic requires the `calc` library.

/tikz/pics/3d/cylinder/axis (initially `(0,0,1)`)

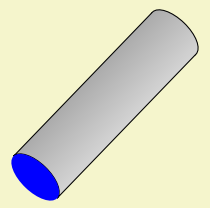
Axis of cylinder.

/tikz/pics/3d/cylinder/mantle (initially `draw`)

Style for cylinder mantle. If no fill option is specified, it will be shaded.

/tikz/pics/3d/cylinder/top (initially `draw`)

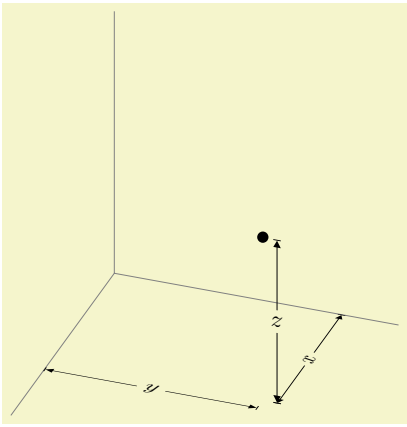
Style for cylinder top.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}[3d/install view={phi=30,psi=0,theta=80}]
 \pic{cylinder={r=0.4,h=3,
 cylinder/axis={(1,0,1)},
 cylinder/top/.style={fill=blue}}};
\end{tikzpicture}
```

/tikz/pics/3d/dim line (initially empty)

Draws a line that allows users to annotate axes and/or indicate dimensions. It expects the user to specify the start and end points, `A` and `B`, as well as the normal `n` of the plane the line is to be drawn. In addition, the `d` key, which can take negative values, determines the distance of the line from `A--B`. The label is given by the `l` key, and the `dim label` style controls its appearance. This `pic` requires the `calc` library.




```
\usetikzlibrary {calc,3dtools}
\begin{tikzpicture}[3d/install view={phi=110,theta=60},
declare function={a=4;rx=2.5;ry=3;rz=2.5;}]
 \draw[gray,help lines] (0,0,0) coordinate (O)
 -- (a,0,0)
(O) -- (0,a,0) (O) -- (0,0,a);
 \path (rx,ry,rz) coordinate[circle,inner sep=1.5pt,
fill] (r);
 \path pic{3d/dim line={A={(0,ry,0)},B={(rx,ry,0)},l={$x$}}}
pic{3d/dim line={A={(rx,0,0)},B={(rx,ry,0)},l={$y$},d=-2mm}}
pic{3d/dim line={A={(rx,ry,0)},B={(r)},n={(1,0,0)},l={$z$},
d=-2mm,dim label/.append style={rotate=90}}};
\end{tikzpicture}
```

To do:

- transform to plane given by three non-degenerate coordinates
- transform to plane given by normal and one point
- maybe layering/visibility

1.6 Path management

This is poor marmots version of path management. Almost certainly spath3 is way superior. The idea is that one can store paths, and redraw them, possibly in reversed order. This recording tool uses the `show path construction` decoration from the `decorations.pathreplacing` library. In order to make use of these tools, you need to load the library.

`/tikz/store path=`⟨*name*⟩ (no default, initially `empty`)

Records a path under the name ⟨*name*⟩. If you want to draw and record a path at the same time, use `postaction={store path=name}`.

`/tikz/stored path/coordinate prefix=`⟨*prefix*⟩ (no default, initially `stored-`)

Prefix for the stored coordinates. If you store several paths and want to possibly combine them, you may want to give each path a unique prefix.

`/tikz/stored path/reset coordinate index` (no value)

Resets the counter for the stored coordinates. Mainly for internal use.

`/tikz/stored path/restore path=`⟨*name*⟩ (no default, initially `empty`)

Restores a saved path of name ⟨*name*⟩ using the `insert path` key.

`/tikz/stored path/restore reversed path=`⟨*name*⟩ (no default, initially `empty`)

Restores a saved path of name ⟨*name*⟩ using the `insert path` key in *reversed* order.

`/tikz/stored path/append path=`⟨*name*⟩ (no default, initially `empty`)

Restores a saved path of name ⟨*name*⟩ using the `insert path` key with a `--` at the beginning. This may be necessary to patch paths together.

`/tikz/stored path/restore reversed path=`⟨*name*⟩ (no default, initially `empty`)

Restores a saved path of name ⟨*name*⟩ using the `insert path` key in *reversed* order with a `--` at the beginning. This may be necessary to patch paths together.

`/tikz/stored path/first coordinate of=`⟨*name*⟩ (no default, initially `empty`)

Inserts the first coordinate of the path named ⟨*name*⟩ using the `insert path` key.

`/tikz/stored path/last coordinate of=`⟨*name*⟩ (no default, initially `empty`)

Inserts the last coordinate of the path named ⟨*name*⟩ using the `insert path` key.

`/tikz/stored path/coordinate with index`=*n* of ⟨*name*⟩ (no default, initially `empty`)

Inserts the n^{th} coordinate of the path named ⟨*name*⟩ using the `insert path` key. Notice that at present there is no sanity check. If *n* is larger than the number of stored coordinates, there will be an error.

`tikztdindexoflastcoordinate`

Yields index of the last coordinate of a stored path. This can be used to test if a path is empty.

1.7 List management

3d ordering has a lot to do with sorting lists. Here are some poor marmot's tools to deal with such lists and some primitive versions of associative arrays. First of all, there are a couple of *key handlers* that allow one to add and remove items. Notice that, in order to apply key handlers to a list, the list has to be stored in a ⟨*key*⟩.

Key handler ⟨*key*⟩`/.add item`

Key handler that adds an item to a list ⟨*key*⟩.

Key handler ⟨*key*⟩`/.remove item`

Key handler that removes an item from a list ⟨*key*⟩.

Key handler ⟨*key*⟩`/.count`=⟨*macro*⟩

Key handler that stores the number of elements of a given list ⟨*key*⟩ in a macro, which is the mandatory argument.

Key handler ⟨*key*⟩`/.print list`

Key handler that prints a list. It threads `/list management/print item` code over the list, and separates the items by `/list management/list separators`.

`/list management/print item` (no value)

Code used by the `.print list` key handler.

`/list management/list separator` (initially `,`)

Code used by the `.print list` key handler to separate items.

The first 9 Ti*k*Zlings are {anteater, bear, bee, cat, coati, hippo, koala, marmot, mouse}.

After the wolpertinger joined, they were 1, and the list became {anteater, bear, bee, cat, coati, hippo, koala, marmot, mouse, wolpertinger}.

Sadly, the mouse could not stand the wolpertinger, so the list Ti*k*Zlings got reduced to 9, consisting of {anteater, bear, bee, cat, coati, hippo, koala, marmot, wolpertinger}.

```
\usetikzlibrary {3dtools}
\begin{minipage}{6.4cm}
\pgfkeys{/my lists/.cd,
tikzlings/.initial={anteater,%
bear,bee,cat,coati,%
hippo,koala,marmot,mouse}}
The first\
\pgfkeys{/my lists/tikzlings/.count=\myn}\myn\
Ti\emph{k}Zlings are\
\{\pgfkeys{/my lists/tikzlings/.print list}\}.
\par\medskip
\pgfkeys{/my lists/tikzlings/.add item=wolpertinger}
After the wolpertinger joined, they were \pgfkeys{/my
lists/tikzlings/.count=\myn}{$\myn$}, and the list became\
\{\pgfkeys{/my lists/tikzlings/.print list}\}.
\par\medskip
\pgfkeys{/my lists/tikzlings/.remove item=mouse}
Sadly, the mouse could not stand the wolpertinger,\
so the list Ti\emph{k}Zlings got reduced to\
\pgfkeys{/my lists/tikzlings/.count=\myn}{$\myn$},\
consisting of\
\{\pgfkeys{/my lists/tikzlings/.print list}\}.
\end{minipage}
```

Key handler ⟨*key*⟩`/.is array`=⟨*list*⟩

Key handler makes the current key ⟨*key*⟩ an array consisting of ⟨*list*⟩. You need to use a new key for that, i.e. you cannot overwrite existing keys. The benefit is that you can access the items in the list simply via ⟨*key*⟩/*n*, where *n* is an integer-valued index, and that ⟨***key***⟩`/dim` contains its dimension.

Key handler ⟨*key*⟩**/.sort numeric list=**⟨*macro*⟩⟨*macro*⟩

Key handler that sorts a list ⟨*key*⟩. The resulting sorted list is stored in the first argument, which has to be a macro, and the permutation is stored in the second argument, which needs to be a macro as well. This allows one to know the position of a given entry in a sorted list.

Often one is given a list of entries where one wants to sort on the basis of some function that can be fed with an entry. One may also want to have an index mappingwhich indicates at which position the i^{th} entry of the initial list ended up getting. In this example, we start out with

$$\{\lambda_i\}_{i=1}^{n} = \{12, 19, 1, 3, 17, 4, 23, 5, 9, 7\} .$$

The fact that we used the `/.is array` key handler means that we know its dimension, $n = 10$. After sorting we get

$$\{\lambda'_i\}_{i=1}^{n} = \{1, 3, 4, 5, 7, 9, 12, 17, 19, 23\}$$

and an index or reshuffling or permutation list

$$P = (3, 4, 6, 8, 10, 9, 1, 5, 2, 7) .$$

For instance, the 5^{th} entry of the permutation list being $P_5 = 10$ tells us that the 10^{th} entry of our original list, $\lambda_{10} = 7$, ended up at position 5 in the sorted list, i.e. $\lambda'_5 = \lambda_{10}$. Similarly, $P_3 = 6$ means that for the 6^{th} entry of our original list, $\lambda_6 = 4 = \lambda'_3$.

```
\usetikzlibrary {3dtools}
\begin{minipage}{6.4cm}
\pgfkeys{/my lists/.cd,
my initial array/.is array={%
12,19,1,3,17,4,23,5,9,7},
my values/.initial=\pgfkeysvalueof{%
/my lists/my initial array/content},%
my values/.sort numeric list=%
{\temp}{\templ},%
my sorted array/.is array/.expanded={\temp},
my index machinery/.is array/.expanded={\templ}}%
Often one is given a list of entries where\
one wants to sort on the basis of some\
function that can be fed with an entry.\
One may also want to have an index mapping
which indicates at which position the\
$i^\mathrm{th}$ entry of the initial list\
ended up getting. In this example, we\
start out with
\[\{\lambda_i\}_{i=1}^n=
\{\pgfkeys{/my lists/my initial array/%
content/.print list}\}
\;.\]
The fact that we used the \texttt{/.is array}\
key handler means that we know its dimension,\
$n=\pgfkeysvalueof{/my lists/my initial array/dim}$.\
After sorting we get
\[\{\lambda_i'\}_{i=1}^n=\{
\pgfkeys{/my lists/my sorted array/%
content/.print list}\}\]
and an index or reshuffling or permutation list
\[P=(\pgfkeys{/my lists/%
my index machinery/content/.print list})\;.\]
For instance, the $5^\mathrm{th}$ entry of the\
permutation list being\
$P_5=\pgfkeysvalueof{/my lists/my index machinery/5}$\
tells us that the\
$\pgfkeysvalueof{/my lists/my index machinery/5}
^\mathrm{th}$\ entry of our original list,\
$\lambda_{\pgfkeysvalueof{/my lists/%
my index machinery/5}}
=\pgfkeysvalueof{/my lists/my initial array/%
\pgfkeysvalueof{/my lists/my index machinery/5}}$,\
ended up at position $5$ in the sorted list,\
i.e.\ $\lambda_5'=\lambda_{\pgfkeysvalueof{/my lists/%
my index machinery/5}}$. Similarly,\
$P_3=\pgfkeysvalueof{/my lists/my index machinery/3}$\
means that for the\
$\pgfkeysvalueof{/my lists/my index machinery/3}
^\mathrm{th}$ entry of our original list,\
$\lambda_{\pgfkeysvalueof{/my lists/%
my index machinery/3}}
=\pgfkeysvalueof{/my lists/my initial array/%
\pgfkeysvalueof{/my lists/my index machinery/3}}
=\lambda_3'$.
\end{minipage}
```

1.8 Miscellaneous goodies

`/tikz/on layer=`⟨*layer*⟩ (no default)

Allows one to set a layer in a path. Taken from https://tex.stackexchange.com/a/20426.

`/tikz/node on layer=`⟨*layer*⟩ (no default)

Allows one to set a layer of a node in a path. Taken from https://tex.stackexchange.com/a/20426.

/tikz/even odd clip (no value)

Enables the ‘even odd rule‘ for clips.

/tikz/save named path (no value)

Saves a path very much like the `save path` key but uses strings, i.e. you can say `save named path=A`, and prefixes the global macro by `tikz@td@named@path@` to minimize the chance that this macro interferes with other macros.

/tikz/use named path (no value)

Allows one to use a path saved with `save named path`, e.g. `use named path=A`.

/tikz/same bounding box=*id* (default `A`)

Allows one to synchronize the bounding box of various pictures. Each picture with the same *id* will have the same bounding box after the second compilation.

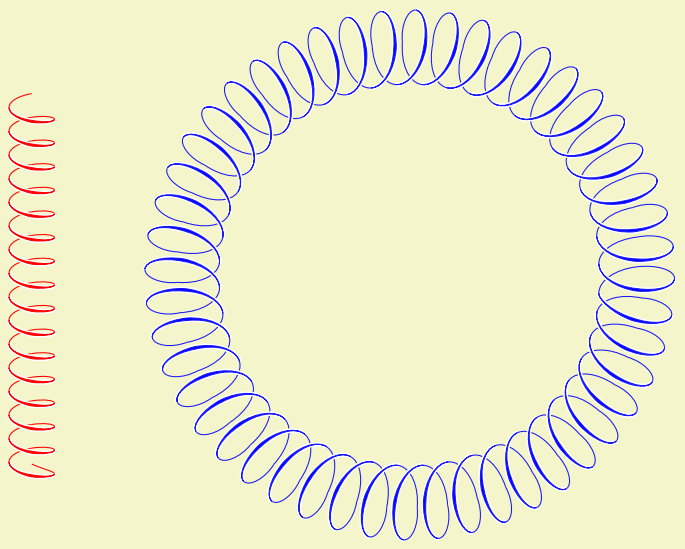
1.9 3D-like decorations

/tikz/decorations/3d complete coil (no value)

3d-like coil where the front is thicker than the back.

/tikz/decorations/3d coil closed (no value)

Indicates that the coil is closed.



```
\usetikzlibrary {3dtools}
\begin{tikzpicture}
\draw[decoration={3d coil color=red,aspect=0.35, segment length=3.1mm,
amplitude=3mm,3d complete  coil},
decorate] (0,1) -- (0,6);
\draw[decoration={3d coil color=blue,3d coil opacity=0.9,aspect=0.5,
segment length={2*pi*3cm/50}, amplitude=5mm,3d complete coil,
3d coil closed},
decorate] (5,3.5) circle[radius=3cm];
\end{tikzpicture}
```

1.10 Remarks on the implementation

When the library gets loaded, the version of Ti*k*Z gets checked. If the version is earlier than 3.1.1, a warning is issued. In this case, one cannot use the library unless one updates the pgf installation. Notice that many features of this library depend on Ti*k*Z's internal macro `\tikz@dcl@coord@`. If this macro gets removed or changed, many features of the library will no longer work. So this whole setup is fragile. Of course, this complication is not unheard of, but it might be important to know about this before deciding to use this library.

Acknowledgments

I'd like to thank minhthien2016 for continuous support and encouragement. I thank the creators, maintainers, moderators and users of topanswers.xyz for their invaluable support. These acknowledgements extend to all users on this site, including those with whom I have passionate disagreements. Disagreements are good, they indicate critical thinking, or in the words of Nobel Laureate Steven Weinberg, physics thrives on contradictions.