The l3draw package
Core drawing support

The LaTeX3 Project*

Released 2018/02/21

1 l3draw documentation

The l3draw package provides a set of tools for creating (vector) drawings in expl3. It is heavily inspired by the pgf layer of the Ti*k*Z system, with many of the interfaces having the same form. However, the code provided here is build entirely on core expl3 ideas and uses the LaTeX3 FPU for numerical support.

Numerical expressions in l3draw are handled as floating point expressions, unless otherwise noted. This means that they may contain or omit explicit units. Where units are omitted, they will automatically be taken as given in (TeX) points.

The code here is *highly* experimental.

1.1 Drawings

`\draw_begin:`
`\draw_end:`

```
\draw_begin:
...
\draw_end:
```

Each drawing should be created within a `\draw_begin:`/`\draw_end:` function pair. The `begin` function sets up a number of key data structures for the rest of the functions here: unless otherwise specified, use of `\draw_...` functions outside of this "environment" is *not supported*.

The drawing created within the environment will be inserted into the typesetting stream by the `\draw_end:` function, which will switch out of vertical mode if required.

1.2 Graphics state

Within the drawing environment, a number of functions control how drawings will appear. Note that these all apply *globally*, though some are rest at the start of each drawing (`\draw_begin:`).

`\g_draw_linewidth_default_dim`

The default value of the linewidth for stokes, set at the start of every drawing (`\draw_begin:`).

*E-mail: latex-team@latex-project.org

`\draw_linewidth:n`
`\draw_inner_linewidth:n`

`\draw_linewidth:n {⟨width⟩}`

Sets the width to be used for stroking to the ⟨*width*⟩ (an ⟨*fp expr*⟩).

`\draw_nonzero_rule:`
`\draw_evenodd_rule:`

`\draw_nonzero_rule:`

Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

`\draw_cap_butt:`
`\draw_cap_rectangle:`
`\draw_cap_round:`

`\draw_cap_butt:`

Sets the style of terminal stroke position to one of butt, rectangle or round.

`\draw_join_bevel:`
`\draw_join_miter:`
`\draw_join_round:`

`\draw_cap_butt:`

Sets the style of stroke joins to one of bevel, miter or round.

`\draw_miterlimit:n`

`\draw_miterlimit:n {⟨limit⟩}`

Sets the miter ⟨*limit*⟩ of lines joined as a miter, as described in the PDF and PostScript manuals. The ⟨*limit*⟩ is an ⟨*fp expr*⟩.

1.3 Points

Functions supporting the calculation of points (co-ordinates) are expandable and may be used outside of the drawing environment. When used in this way, they all yield a co-ordinate tuple, for example

```
\tl_set:Nx \l_tmpa_tl { \draw_point:nn { 1 } { 2 } }
\tl_show:N \l_tmpa_tl
```

gives

```
> \l_tmpa_tl=1pt,2pt.
<recently read> }
```

This output form is then suitable as *input* for subsequent point calculations, *i.e.* where a ⟨*point*⟩ is required it may be given as a tuple. This *may* include units and surrounding parentheses, for example

```
1,2
(1,2)
1cm,3pt
(1pt,2cm)
2 * sind(30), 2^4in
```

are all valid input forms. Notice that each part of the tuple may itself be a float point expression.

Point co-ordinates are relative to the canvas axes, but can be transformed by `\draw_point_transform:n`. These manipulation is applied by many higher-level functions, for example path construction, and allows parts of a drawing to be rotated, scaled or skewed. This occurs before writing any data to the driver, and so such manipulations are tracked by the drawing mechanisms. See `\driver_draw_transformcm:nnnnnn` for driver-level manipulation of the canvas axes themselves.

Notice that in contrast to `pgf` it is possible to give the positions of points *directly*.

1.3.1 Basic point functions

\draw_point:nn ⋆ `\draw_point:nn {⟨x⟩} {⟨y⟩}`

Gives the co-ordinates of the point at ⟨*x*⟩ and ⟨*y*⟩, both of which are ⟨*fp expr*⟩.

\draw_point_polar:nn ⋆ `\draw_point_polar:nn {⟨angle⟩} {⟨radius⟩}`
\draw_point_polar:nnn ⋆ `\draw_point_polar:nnn {⟨angle⟩} {⟨radius-a⟩} {⟨radius-b⟩}`

Gives the co-ordinates of the point at ⟨*angle*⟩ (an ⟨*fp expr*⟩ in *degrees*) and ⟨*radius*⟩. The three-argument version accepts two radii of different lengths.

Note the interface here is somewhat different from that in pgf: the one- and two-radii versions in l3draw use separate functions, whilst in pgf they use the same function and a keyword.

\draw_point_add:nn ⋆ `\draw_point_add:nn {⟨point1⟩} {⟨point2⟩}`

Adds ⟨*point1*⟩ to ⟨*point2*⟩.

\draw_point_diff:nn ⋆ `\draw_point_diff:nn {⟨point1⟩} {⟨point2⟩}`

Subtracts ⟨*point1*⟩ from ⟨*point2*⟩.

\draw_point_scale:nn ⋆ `\draw_point_scale:nn {⟨scale⟩} {⟨point⟩}`

Scales the ⟨*point*⟩ by the ⟨*scale*⟩ (an ⟨*fp expr*⟩).

\draw_point_unit_vector:n ⋆ `\draw_point_unit_vector:n {⟨point⟩}`

Expands to the co-ordinates of a unit vector joining the ⟨*point*⟩ with the origin.

\draw_point_transform:n ⋆ `\draw_point_transform:n {⟨point⟩}`

Evaluates the position of the ⟨*point*⟩ subject to the current transformation matrix. This operation is applied automatically by most higher-level functions (*e.g.* path manipulations).

1.3.2 Points on a vector basis

As well as giving explicit values, it is possible to describe points in terms of underlying direction vectors. The latter are initially co-incident with the standard Cartesian axes, but may be altered by the user.

\draw_xvec_set:n `\draw_xvec_set:n {⟨point⟩}`
\draw_yvec_set:n
\draw_zvec_set:n

Defines the appropriate base vector to point toward the ⟨*point*⟩ on the canvas. The standard settings for the x- and y-vectors are 1 cm along the relevant canvas axis, whilst for the z-vector an appropriate direction is taken.

\draw_point_vec:nn ⋆ `\draw_point_vec:nn {⟨xscale⟩} {⟨yscale⟩}`
\draw_point_vec:nnn ⋆ `\draw_point_vec:nnn {⟨xscale⟩} {⟨yscale⟩} {⟨zscale⟩}`

Expands to the co-ordinate of the point at ⟨*xscale*⟩ times the x-vector and ⟨*yscale*⟩ times the y-vector. The three-argument version extends this to include the z-vector.

\draw_point_vec_polar:nn ⋆ \draw_point_vec_polar:nn {⟨angle⟩} {⟨radius⟩}
\draw_point_vec_polar:nnn ⋆ \draw_point_vec_polar:nnn {⟨angle⟩} {⟨radius-a⟩} {⟨radius-b⟩}

Gives the co-ordinates of the point at ⟨*angle*⟩ (an ⟨*fp expr*⟩ in *degrees*) and ⟨*radius*⟩, relative to the prevailing x- and y-vectors. The three-argument version accepts two radii of different lengths.

Note the interface here is somewhat different from that in pgf: the one- and two-radii versions in l3draw use separate functions, whilst in pgf they use the same function and a keyword.

1.3.3 Intersections

\draw_point_intersect_lines:nnnn ⋆ \draw_point_intersect_lines:nnnn {⟨*point1*⟩} {⟨*point2*⟩} {⟨*point3*⟩} {⟨*point4*⟩}

Evaluates the point at the intersection of one line, joining ⟨*point1*⟩ and ⟨*point2*⟩, and a second line joining ⟨*point3*⟩ and ⟨*point4*⟩. If the lines do not intersect, or are coincident, and error will occur.

\draw_point_intersect_circles:nnnn ⋆ \draw_point_intersect_circles:nnnnn {⟨*center1*⟩} {⟨*radius1*⟩} {⟨*center2*⟩} {⟨*radius2*⟩} {⟨*root*⟩}

Evaluates the point at the intersection of one circle with ⟨*center1*⟩ and ⟨*radius1*⟩, and a second circle with ⟨*center2*⟩ and ⟨*radius2*⟩. If the circles do not intersect, or are coincident, and error will occur.

Note the interface here has a different argument ordering from that in pgf, which has the two centers then the two radii.

1.3.4 Interpolations

\draw_point_interpolate_line:nnn ⋆ \draw_point_interpolate_line:nnn {⟨*part*⟩} {⟨*point1*⟩} {⟨*point2*⟩}

Expands to the point which is ⟨*part*⟩ way along the line joining ⟨*point1*⟩ and ⟨*point2*⟩. The ⟨*part*⟩ may be an interpolation or an extrapolation, and is a floating point value expressing a percentage along the line, *e.g.* a value of `0.5` would be half-way between the two points.

\draw_point_interpolate_distance:nnn ⋆ \draw_point_interpolate_distance:nnn {⟨*distance*⟩} {⟨*point expr1*⟩} {⟨*point expr2*⟩}

Expands to the point which is ⟨*distance*⟩ way along the line joining ⟨*point1*⟩ and ⟨*point2*⟩. The ⟨*distance*⟩ may be an interpolation or an extrapolation.

\draw_point_interpolate_curve:nnnnnn ⋆ \draw_point_interpolate_curve:nnnnnn {⟨*part*⟩} {⟨*start*⟩} {⟨*control1*⟩} {⟨*control2*⟩} {⟨*end*⟩}

Expands to the point which is ⟨*part*⟩ way along the curve between ⟨*start*⟩ and ⟨*end*⟩ and defined by ⟨*control1*⟩ and ⟨*control2*⟩. The ⟨*part*⟩ may be an interpolation or an extrapolation, and is a floating point value expressing a percentage along the curve, *e.g.* a value of `0.5` would be half-way along the curve.

1.4 Paths

Paths are constructed by combining one or more operations before applying one or more actions. Thus until a path is "used", it may be manipulated or indeed discarded entirely. Only one path is active at any one time, and the path is *not* affected by TEX grouping.

`\draw_path_corner_arc:n` `\draw_path_corner_arc:n {⟨length⟩}`

Sets the degree of rounding applied to corners in a path: if the ⟨*length*⟩ is `0pt` then no rounding applies. The value of the ⟨*length*⟩ is local to the current TEX group. *At present, corner arcs are not activated in the code.*

`\draw_path_moveto:n` `\draw_path_moveto:n {⟨point⟩}`

Moves the reference point of the path to the ⟨*point*⟩, but will not join this to any previous point.

`\draw_path_lineto:n` `\draw_path_lineto:n {⟨point⟩}`

Joins the current path to the ⟨*point*⟩ with a straight line.

`\draw_path_curveto:nnn` `\draw_path_curveto:nnn {⟨control1⟩} {⟨control2⟩} {⟨end⟩}`

Joins the current path to the ⟨*end*⟩ with a curved line defined by cubic BÃľzier points ⟨*control1*⟩ and ⟨*control2*⟩.

`\draw_path_curveto:nn` `\draw_path_curveto:nn {⟨control⟩} {⟨end⟩}`

Joins the current path to the ⟨*end*⟩ with a curved line defined by quadratic BÃľzier point ⟨*control*⟩.

`\draw_path_arc:nnn` `\draw_path_arc:nnn {⟨angle1⟩} {⟨angle2⟩} {⟨radius⟩}`
`\draw_path_arc:nnnn` `\draw_path_arc:nnnn {⟨angle1⟩} {⟨angle2⟩} {⟨radius-a⟩} {⟨radius-b⟩}`

Joins the current path with an arc between ⟨*angle1*⟩ and ⟨*angle2*⟩ and of ⟨*radius*⟩. The four-argument version accepts two radii of different lengths.

Note the interface here has a different argument ordering from that in pgf, which has the two centers then the two radii.

`\draw_path_arc_axes:nnnn` `\draw_path_arc_axes:nnn {⟨angle1⟩} {⟨angle2⟩} {⟨vector1⟩} {⟨vector2⟩}`

Appends the portion of an ellipse from ⟨*angle1*⟩ to ⟨*angle2*⟩ of an ellipse with axes along ⟨*vector1*⟩ and ⟨*vector2*⟩ to the current path.

`\draw_path_ellipse:nnnn` `\draw_path_ellipse:nnn {⟨center⟩} {⟨vector1⟩} {⟨vector2⟩}`

Appends an ellipse at ⟨*center*⟩ with axes along ⟨*vector1*⟩ and ⟨*vector2*⟩ to the current path.

`\draw_path_circle:nn` `\draw_path_circle:nn {⟨center⟩} {⟨radius⟩}`

Appends a circle of ⟨*radius*⟩ at ⟨*center*⟩ to the current path.

\draw_path_rectangle:nn
\draw_path_rectangle_corners:nn

\draw_path_rectangle:nn {⟨*lower-left*⟩} {⟨*displacement*⟩}
\draw_path_rectangle_corners:nn {⟨*lower-left*⟩} {⟨*top-right*⟩}

Appends a rectangle starting at ⟨*lower-left*⟩ to the current path, with the size of the rectangle determined either by a ⟨*displacement*⟩ or the position of the ⟨*top-right*⟩.

\draw_path_grid:nnnn

\draw_path_grid:nnnn {⟨*xspace*⟩} {⟨*yspace*⟩} {⟨*lower-left*⟩} {⟨*upper-right*⟩}

Constructs a grid of ⟨*xspace*⟩ and ⟨*yspace*⟩ from the ⟨*lower-left*⟩ to the ⟨*upper-right*⟩, and appends this to the current path.

\draw_path_close:

\draw_path_close:

Closes the current part of the path by appending a straight line from the current point to the starting point of the path.

\draw_path_use:n
\draw_path_use_clear:n

\draw_path_use:n {⟨*action(s)*⟩}

Inserts the current path, carrying out one ore more possible ⟨*actions*⟩ (a comma list):

- `clear` Resets the path to empty
- `clip` Clips any content outside of the path
- `draw`
- `fill` Fills the interior of the path with the current file color
- `stroke` Draws a line along the current path

1.5 Color

\draw_color:n
\draw_fill:n
\draw_stroke:n

\draw_color:n {⟨*color expression*⟩}

Evaluates the ⟨*color expression*⟩ as described for l3color.

1.6 Transformations

Points are normally used unchanged relative to the canvas axes. This can be modified by applying a transformation matrix. The canvas axes themselves may be adjusted using `\driver_draw_transformcm:nnnnnn`: note that this is transparent to the drawing code so is not tracked.

\draw_transform_reset:

\draw_transform_reset:

Resets the matrix to the identity.

\draw_transform_concat:nnnnn

\draw_transform_concat:nnnnn
{⟨a⟩} {⟨*b*⟩} {⟨*c*⟩} {⟨*d*⟩} {⟨*vector*⟩}

Appends the given transformation to the currently-active one. The transformation is made up of a matrix ⟨*a*⟩, ⟨*b*⟩, ⟨*c*⟩ and ⟨*d*⟩, and a shift by the ⟨*vector*⟩.

`\draw_transform:nnnnn` `\draw_transform:nnnnn {⟨a⟩} {⟨b⟩} {⟨c⟩} {⟨d⟩} {⟨vector⟩}`

Applies the transformation matrix specified, over-writing any existing matrix. The transformation is made up of a matrix $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$ and $\langle d \rangle$, and a shift by the $\langle vector \rangle$.

`\draw_transform_triangle:nnn` `\draw_transform_triangle:nnn {⟨origin⟩} {⟨point1⟩} {⟨point2⟩}`

Applies a transformation such that the co-ordinates $(0, 0)$, $(1, 0)$ and $(0, 1)$ are given by the $\langle origin \rangle$, $\langle point1 \rangle$ and $\langle point2 \rangle$, respectively.

`\draw_transform_invert:` `\draw_transform_invert:`

Inverts the current transformation matrix and reverses the current shift vector.

File I
Implementation

1 l3draw implementation

```
1 ⟨*initex | package⟩
2 ⟨@@=draw⟩
3 ⟨*package⟩
4 \ProvidesExplPackage{l3draw}{2018/02/21}{}
5   {L3 Experimental core drawing support}
6 ⟨/package⟩
7 \RequirePackage { l3color }
```

Everything else is in the sub-files!

```
8 ⟨/initex | package⟩
```

2 l3draw-paths implementation

```
9 ⟨*initex | package⟩
10 ⟨@@=draw⟩
```

This sub-module covers more-or-less the same ideas as `pgfcorepathconstruct.code.tex`, though using the expandable FPU means that the implementation often varies. At present, equivalents of the following are currently absent:

- `\pgfpatharcto`, `\pgfpatharctoprecomputed`: These are extremely specialised and are very complex in implementation. If the functionality is required, it is likely that it will be set up from scratch here.
- `\pgfpathparabola`: Seems to be unused other than defining a Ti*k*Z interface, which itself is then not used further.
- `\pgfpathsine`, `\pgfpathcosine`: Need to see exactly how these need to work, in particular whether a wider input range is needed and what approximation to make.
- `\pgfpathcurvebetweentime`, `\pgfpathcurvebetweentimecontinue`: These don't seem to be used at all.

\l__draw_path_tmp_tl
\l__draw_path_tmpa_fp
\l__draw_path_tmpb_fp

Scratch space.

```
\tl_new:N \l__draw_path_tmp_tl
\fp_new:N \l__draw_path_tmpa_fp
\fp_new:N \l__draw_path_tmpb_fp
```

(*End definition for* `\l__draw_path_tmp_tl`, `\l__draw_path_tmpa_fp`, *and* `\l__draw_path_tmpb_fp`.)

2.1 Tracking paths

\g__draw_path_lastx_dim
\g__draw_path_lasty_dim

The last point visited on a path.

```
\dim_new:N \g__draw_path_lastx_dim
\dim_new:N \g__draw_path_lasty_dim
```

(*End definition for* `\g__draw_path_lastx_dim` *and* `\g__draw_path_lasty_dim`.)

\g__draw_path_xmax_dim
\g__draw_path_xmin_dim
\g__draw_path_ymax_dim
\g__draw_path_ymin_dim

The limiting size of a path.

```
\dim_new:N \g__draw_path_xmax_dim
\dim_new:N \g__draw_path_xmin_dim
\dim_new:N \g__draw_path_ymax_dim
\dim_new:N \g__draw_path_ymin_dim
```

(*End definition for* `\g__draw_path_xmax_dim` *and others.*)

__draw_path_update_limits:nn
__draw_path_reset_limits:

Track the limits of a path and (perhaps) of the picture as a whole. (At present the latter is always true: that will change as more complex functionality is added.)

```
\cs_new_protected:Npn \__draw_path_update_limits:nn #1#2
  {
    \dim_gset:Nn \g__draw_path_xmax_dim
      { \dim_max:nn \g__draw_path_xmax_dim {#1} }
    \dim_gset:Nn \g__draw_path_xmin_dim
      { \dim_min:nn \g__draw_path_xmin_dim {#1} }
    \dim_gset:Nn \g__draw_path_ymax_dim
      { \dim_max:nn \g__draw_path_ymax_dim {#2} }
    \dim_gset:Nn \g__draw_path_ymin_dim
      { \dim_min:nn \g__draw_path_ymin_dim {#2} }
    \bool_if:NT \l__draw_update_bb_bool
      {
        \dim_gset:Nn \g__draw_xmax_dim
          { \dim_max:nn \g__draw_xmax_dim {#1} }
        \dim_gset:Nn \g__draw_xmin_dim
          { \dim_min:nn \g__draw_xmin_dim {#1} }
        \dim_gset:Nn \g__draw_ymax_dim
          { \dim_max:nn \g__draw_ymax_dim {#2} }
        \dim_gset:Nn \g__draw_ymin_dim
          { \dim_min:nn \g__draw_ymin_dim {#2} }
      }
  }
\cs_new_protected:Npn \__draw_path_reset_limits:
  {
    \dim_gset:Nn \g__draw_path_xmax_dim { -\c_max_dim }
    \dim_gset:Nn \g__draw_path_xmin_dim {  \c_max_dim }
    \dim_gset:Nn \g__draw_path_ymax_dim { -\c_max_dim }
    \dim_gset:Nn \g__draw_path_ymin_dim {  \c_max_dim }
  }
```

(*End definition for* `\__draw_path_update_limits:nn` *and* `\__draw_path_reset_limits:`.)

`\__draw_path_update_last:nn` A simple auxiliary to avoid repetition.

```
\cs_new_protected:Npn \__draw_path_update_last:nn #1#2
  {
    \dim_gset:Nn \g__draw_path_lastx_dim {#1}
    \dim_gset:Nn \g__draw_path_lasty_dim {#2}
  }
```

(*End definition for* `\__draw_path_update_last:nn`.)

2.2 Corner arcs

At the level of path *construction*, rounded corners are handled by inserting a marker into the path: that is then picked up once the full path is constructed. Thus we need to set up the appropriate data structures here, such that this can be applied every time it is relevant.

`\l__draw_corner_xarc_dim` `\l__draw_corner_yarc_dim` The two arcs in use.

```
\dim_new:N \l__draw_corner_xarc_dim
\dim_new:N \l__draw_corner_yarc_dim
```

(*End definition for* `\l__draw_corner_xarc_dim` *and* `\l__draw_corner_yarc_dim`.)

`\l__draw_corner_arc_bool` A flag to speed up the repeated checks.

```
\bool_new:N \l__draw_corner_arc_bool
```

(*End definition for* `\l__draw_corner_arc_bool`.)

`\draw_path_corner_arc:n` `\__draw_path_corner_arc:nn` Calculate the arcs, check they are non-zero.

```
\cs_new_protected:Npn \draw_path_corner_arc:n #1
  {
    \__draw_point_process:nn { \__draw_path_corner_arc:nn } {#1}
  }
\cs_new_protected:Npn \__draw_path_corner_arc:nn #1#2
  {
    \dim_set:Nn \l__draw_corner_xarc_dim {#1}
    \dim_set:Nn \l__draw_corner_yarc_dim {#2}
    \bool_lazy_and:nnTF
      { \dim_compare_p:nNn \l__draw_corner_xarc_dim = { 0pt } }
      { \dim_compare_p:nNn \l__draw_corner_yarc_dim = { 0pt } }
      { \bool_set_false:N \l__draw_corner_arc_bool }
      { \bool_set_true:N \l__draw_corner_arc_bool }
  }
```

(*End definition for* `\draw_path_corner_arc:n` *and* `\__draw_path_corner_arc:nn`. *This function is documented on page* 5.)

`\__draw_path_mark_corner:` Mark up corners for arc post-processing.

```
\cs_new_protected:Npn \__draw_path_mark_corner:
  {
    \bool_if:NT \l__draw_corner_arc_bool
      {
        \__draw_softpath_roundpoint:VV
```

```
          \l__draw_corner_xarc_dim
          \l__draw_corner_yarc_dim
      }
  }
```

(*End definition for* `\__draw_path_mark_corner:`.)

2.3 Basic path constructions

`\draw_path_moveto:n`
`\draw_path_lineto:n`
`\__draw_path_moveto:nn`
`\__draw_path_lineto:nn`
`\draw_path_curveto:nnn`
`\__draw_path_curveto:nnnnnn`

At present, stick to purely linear transformation support and skip the soft path business: that will likely need to be revisited later.

```
\cs_new_protected:Npn \draw_path_moveto:n #1
  {
    \__draw_point_process:nn
      { \__draw_path_moveto:nn }
      { \draw_point_transform:n {#1} }
  }
\cs_new_protected:Npn \__draw_path_moveto:nn #1#2
  {
     \__draw_path_update_limits:nn {#1} {#2}
     \__draw_softpath_moveto:nn {#1} {#2}
     \__draw_path_update_last:nn {#1} {#2}
  }
\cs_new_protected:Npn \draw_path_lineto:n #1
  {
    \__draw_point_process:nn
      { \__draw_path_lineto:nn }
      { \draw_point_transform:n {#1} }
  }
\cs_new_protected:Npn \__draw_path_lineto:nn #1#2
  {
     \__draw_path_mark_corner:
     \__draw_path_update_limits:nn {#1} {#2}
     \__draw_softpath_lineto:nn {#1} {#2}
     \__draw_path_update_last:nn {#1} {#2}
  }
\cs_new_protected:Npn \draw_path_curveto:nnn #1#2#3
  {
    \__draw_point_process:nnn
      {
        \__draw_point_process:nn
          {
            \__draw_path_mark_corner:
            \__draw_path_curveto:nnnnnn
          }
          { \draw_point_transform:n {#1} }
      }
      { \draw_point_transform:n {#2} }
      { \draw_point_transform:n {#3} }
  }
\cs_new_protected:Npn \__draw_path_curveto:nnnnnn #1#2#3#4#5#6
  {
     \__draw_path_update_limits:nn {#1} {#2}
     \__draw_path_update_limits:nn {#3} {#4}
```

```
    \__draw_path_update_limits:nn {#5} {#6}
    \__draw_softpath_curveto:nnnnnn {#1} {#2} {#3} {#4} {#5} {#6}
    \__draw_path_update_last:nn {#5} {#6}
  }
```

(*End definition for* `\draw_path_moveto:n` *and others. These functions are documented on page 5.*)

`\draw_path_close:` A simple wrapper.

```
\cs_new_protected:Npn \draw_path_close:
  {
    \__draw_path_mark_corner:
    \__draw_softpath_closepath:
  }
```

(*End definition for* `\draw_path_close:`*. This function is documented on page 6.*)

2.4 Computed curves

More complex operations need some calculations. To assist with those, various constants are pre-defined.

`\draw_path_curveto:nn` `\__draw_path_curveto:nnnn` `\c__draw_path_curveto_a_fp` `\c__draw_path_curveto_b_fp`

A quadratic curve with one control point (x_c, y_c). The two required control points are then

$$x_1 = \frac{1}{3}x_s + \frac{2}{3}x_c \quad y_1 = \frac{1}{3}y_s + \frac{2}{3}y_c$$

and

$$x_2 = \frac{1}{3}x_e + \frac{2}{3}x_c \quad x_2 = \frac{1}{3}y_e + \frac{2}{3}y_c$$

using the start (last) point (x_s, y_s) and the end point (x_s, y_s).

```
\cs_new_protected:Npn \draw_path_curveto:nn #1#2
  {
    \__draw_point_process:nnn
      { \__draw_path_curveto:nnnn }
      { \draw_point_transform:n {#1} }
      { \draw_point_transform:n {#2} }
  }
\cs_new_protected:Npn \__draw_path_curveto:nnnn #1#2#3#4
  {
    \fp_set:Nn \l__draw_path_tmpa_fp { \c__draw_path_curveto_b_fp * #1 }
    \fp_set:Nn \l__draw_path_tmpb_fp { \c__draw_path_curveto_b_fp * #2 }
    \use:x
      {
        \__draw_path_mark_corner:
        \__draw_path_curveto:nnnnnn
          {
            \fp_to_dim:n
              {
                  \c__draw_path_curveto_a_fp * \g__draw_path_lastx_dim
                + \l__draw_path_tmpa_fp
              }
          }
          {
            \fp_to_dim:n
              {
```

```
                  \c__draw_path_curveto_a_fp * \g__draw_path_lasty_dim
                + \l__draw_path_tmpb_fp
              }
          }
          {
            \fp_to_dim:n
              { \c__draw_path_curveto_a_fp * #3 + \l__draw_path_tmpa_fp }
          }
          {
            \fp_to_dim:n
              { \c__draw_path_curveto_a_fp * #4 + \l__draw_path_tmpb_fp }
          }
          {#3}
          {#4}
      }
  }
\fp_const:Nn \c__draw_path_curveto_a_fp { 1 / 3 }
\fp_const:Nn \c__draw_path_curveto_b_fp { 2 / 3 }
```

(*End definition for* `\draw_path_curveto:nn` *and others. This function is documented on page* 5.)

\draw_path_arc:nnn
\draw_path_arc:nnnn
__draw_path_arc:nnnn
__draw_path_arc:nnNnn
__draw_path_arc_auxi:nnnnNnn
__draw_path_arc_auxi:fnnnNnn
__draw_path_arc_auxi:fnfnNnn
__draw_path_arc_auxii:nnnNnnnn
__draw_path_arc_auxiii:nn
__draw_path_arc_auxiv:nnnn
__draw_path_arc_auxv:nn
__draw_path_arc_auxvi:nn
__draw_path_arc_add:nnnn
\l__draw_path_arc_delta_fp
\l__draw_path_arc_start_fp
\c__draw_path_arc_90_fp
\c__draw_path_arc_60_fp

Drawing an arc means dividing the total curve required into sections: using BÃlzier curves we can cover at most 90° at once. To allow for later manipulations, we aim to have roughly equal last segments to the line, with the split set at a final part of 115°.

```
\cs_new_protected:Npn \draw_path_arc:nnn #1#2#3
  { \draw_path_arc:nnnn {#1} {#2} {#3} {#3} }
\cs_new_protected:Npn \draw_path_arc:nnnn #1#2#3#4
  {
    \use:x
      {
        \__draw_path_arc:nnnn
          { \fp_eval:n {#1} }
          { \fp_eval:n {#2} }
          { \fp_to_dim:n {#3} }
          { \fp_to_dim:n {#4} }
      }
  }
\cs_new_protected:Npn \__draw_path_arc:nnnn #1#2#3#4
  {
    \fp_compare:nNnTF {#1} > {#2}
      { \__draw_path_arc:nnNnn {#1} {#2} - {#3} {#4} }
      { \__draw_path_arc:nnNnn {#1} {#2} + {#3} {#4} }
  }
\cs_new_protected:Npn \__draw_path_arc:nnNnn #1#2#3#4#5
  {
    \fp_set:Nn \l__draw_path_arc_start_fp {#1}
    \fp_set:Nn \l__draw_path_arc_delta_fp { abs( #1 - #2 ) }
    \fp_while_do:nNnn { \l__draw_path_arc_delta_fp } > { 90 }
      {
        \fp_compare:nNnTF \l__draw_path_arc_delta_fp > { 115 }
          {
            \__draw_path_arc_auxi:ffnnNnn
              { \fp_to_decimal:N \l__draw_path_arc_start_fp }
              { \fp_eval:n { \l__draw_path_arc_start_fp #3 90 } }
```

```
              { 90 } {#2}
              #3 {#4} {#5}
          }
          {
            \__draw_path_arc_auxi:ffnnNnn
              { \fp_to_decimal:N \l__draw_path_arc_start_fp }
              { \fp_eval:n { \l__draw_path_arc_start_fp #3 60 } }
              { 60 } {#2}
              #3 {#4} {#5}
          }
      }
    \__draw_path_mark_corner:
    \__draw_path_arc_auxi:fnfnNnn
      { \fp_to_decimal:N \l__draw_path_arc_start_fp }
      {#2}
      { \fp_eval:n { abs( \l__draw_path_arc_start_fp - #2 ) } }
      {#2}
      #3 {#4} {#5}
  }
```

The auxiliary is responsible for calculating the required points. The "magic" number required to determine the length of the control vectors is well-established for a right-angle: $\frac{4}{3}(\sqrt{2}-1) = 0.552\,284\,75$. For other cases, we follow the calculation used by pgf but with the second common case of 60° pre-calculated for speed.

```
\cs_new_protected:Npn \__draw_path_arc_auxi:nnnnNnn #1#2#3#4#5#6#7
  {
    \use:x
      {
        \__draw_path_arc_auxii:nnnNnnnn
          {#1} {#2} {#4} #5 {#6} {#7}
          {
            \fp_to_dim:n
              {
                \cs_if_exist_use:cF
                  { c__draw_path_arc_ #3 _fp }
                  { 4/3 * tand( 0.25 * #3 ) }
                  * #6
              }
          }
          {
            \fp_to_dim:n
              {
                \cs_if_exist_use:cF
                  { c__draw_path_arc_ #3 _fp }
                  { 4/3 * tand( 0.25 * #3 ) }
                  * #7
              }
          }
      }
  }
\cs_generate_variant:Nn \__draw_path_arc_auxi:nnnnNnn { fnf , ff }
```

We can now calculate the required points. As everything here is non-expandable, that is best done by using x-type expansion to build up the tokens. The three points are calculated out-of-order, since finding the second control point needs the position of the

end point. Once the points are found, fire-off the fundamental path operation and update the record of where we are up to. The final point has to be

```
\cs_new_protected:Npn \__draw_path_arc_auxii:nnnNnnnn #1#2#3#4#5#6#7#8
  {
    \tl_clear:N \l__draw_path_tmp_tl
    \__draw_point_process:nn
      { \__draw_path_arc_auxiii:nn }
      {
        \__draw_point_transform_noshift:n
          { \draw_point_polar:nnn { #1 #4 90 } {#7} {#8} }
      }
    \__draw_point_process:nn
      {
        \__draw_point_process:nn
          { \__draw_path_arc_auxiv:nnnn }
          {
            \draw_point_transform:n
              { \draw_point_polar:nnn {#1} {#5} {#6} }
          }
      }
      {
        \draw_point_transform:n
          { \draw_point_polar:nnn {#2} {#5} {#6} }
      }
    \__draw_point_process:nn
      { \__draw_path_arc_auxv:nn }
      {
        \__draw_point_transform_noshift:n
          { \draw_point_polar:nnn { #2 #4 -90 } {#7} {#8} }
      }
    \exp_after:wN \__draw_path_curveto:nnnnnn \l__draw_path_tmp_tl
    \fp_set:Nn \l__draw_path_arc_delta_fp { abs ( #2 - #3 ) }
    \fp_set:Nn \l__draw_path_arc_start_fp {#2}
  }
```

The first control point.

```
\cs_new_protected:Npn \__draw_path_arc_auxiii:nn #1#2
  {
    \__draw_path_arc_aux_add:nn
      { \g__draw_path_lastx_dim + #1 }
      { \g__draw_path_lasty_dim + #2 }
  }
```

The end point: simple arithmetic.

```
\cs_new_protected:Npn \__draw_path_arc_auxiv:nnnn #1#2#3#4
  {
    \__draw_path_arc_aux_add:nn
      { \g__draw_path_lastx_dim - #1 + #3 }
      { \g__draw_path_lasty_dim - #2 + #4 }
  }
```

The second control point: extract the last point, do some rearrangement and record.

```
\cs_new_protected:Npn \__draw_path_arc_auxv:nn #1#2
  {
    \exp_after:wN \__draw_path_arc_auxvi:nn
```

```
      \l__draw_path_tmp_tl {#1} {#2}
  }
\cs_new_protected:Npn \__draw_path_arc_auxvi:nn #1#2#3#4#5#6
  {
    \tl_set:Nn \l__draw_path_tmp_tl { {#1} {#2} }
    \__draw_path_arc_aux_add:nn
      { #5 + #3 }
      { #6 + #4 }
    \tl_put_right:Nn \l__draw_path_tmp_tl { {#3} {#4} }
  }
\cs_new_protected:Npn \__draw_path_arc_aux_add:nn #1#2
  {
    \tl_put_right:Nx \l__draw_path_tmp_tl
      { { \fp_to_dim:n {#1} } { \fp_to_dim:n {#2} } }
  }
\fp_new:N \l__draw_path_arc_delta_fp
\fp_new:N \l__draw_path_arc_start_fp
\fp_const:cn { c__draw_path_arc_90_fp } { 4/3 * (sqrt(2) - 1) }
\fp_const:cn { c__draw_path_arc_60_fp } { 4/3 * tand(15) }
```

(*End definition for* `\draw_path_arc:nnn` *and others. These functions are documented on page* 5.)

`\draw_path_arc_axes:nnnn` A simple wrapper.

```
\cs_new_protected:Npn \draw_path_arc_axes:nnnn #1#2#3#4
  {
    \draw_transform_triangle:nnn { 0cm , 0cm } {#3} {#4}
    \draw_path_arc:nnn {#1} {#2} { 1pt }
  }
```

(*End definition for* `\draw_path_arc_axes:nnnn`. *This function is documented on page* 5.)

`\draw_path_ellipse:nnn`
`\__draw_path_ellipse:nnnnnn`
`\__draw_path_ellipse_arci:nnnnnn`
`\__draw_path_ellipse_arcii:nnnnnn`
`\__draw_path_ellipse_arciii:nnnnnn`
`\__draw_path_ellipse_arciv:nnnnnn`
`\c__draw_path_ellipse_fp`

Drawing an ellipse is an optimised version of drawing an arc, in particular reusing the same constant. We need to deal with the ellipse in four parts and also deal with moving to the right place, closing it and ending up back at the center. That is handled on a per-arc basis, each in a separate auxiliary for readability.

```
\cs_new_protected:Npn \draw_path_ellipse:nnn #1#2#3
  {
    \__draw_point_process:nnn
      {
        \__draw_point_process:nn
          { \__draw_path_ellipse:nnnnnn }
          { \draw_point_transform:n {#1} }
      }
      { \__draw_point_transform_noshift:n {#2} }
      { \__draw_point_transform_noshift:n {#3} }
  }
\cs_new_protected:Npn \__draw_path_ellipse:nnnnnn #1#2#3#4#5#6
  {
    \use:x
      {
        \__draw_path_moveto:nn
          { \fp_to_dim:n { #1 + #3 } } { \fp_to_dim:n { #2 + #4 } }
        \__draw_path_ellipse_arci:nnnnnn   {#1} {#2} {#3} {#4} {#5} {#6}
        \__draw_path_ellipse_arcii:nnnnnn  {#1} {#2} {#3} {#4} {#5} {#6}
```

```
        \__draw_path_ellipse_arciii:nnnnnn {#1} {#2} {#3} {#4} {#5} {#6}
        \__draw_path_ellipse_arciv:nnnnnn  {#1} {#2} {#3} {#4} {#5} {#6}
      }
    \__draw_softpath_closepath:
    \__draw_path_moveto:nn {#1} {#2}
  }
\cs_new:Npn \__draw_path_ellipse_arci:nnnnnn #1#2#3#4#5#6
  {
    \__draw_path_curveto:nnnnnn
      { \fp_to_dim:n { #1 + #3 + #5 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #2 + #4 + #6 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #1 + #3 * \c__draw_path_ellipse_fp + #5 } }
      { \fp_to_dim:n { #2 + #4 * \c__draw_path_ellipse_fp + #6 } }
      { \fp_to_dim:n { #1 + #5 } }
      { \fp_to_dim:n { #2 + #6 } }
  }
\cs_new:Npn \__draw_path_ellipse_arcii:nnnnnn #1#2#3#4#5#6
  {
    \__draw_path_curveto:nnnnnn
      { \fp_to_dim:n { #1 - #3 * \c__draw_path_ellipse_fp + #5 } }
      { \fp_to_dim:n { #2 - #4 * \c__draw_path_ellipse_fp + #6 } }
      { \fp_to_dim:n { #1 - #3 + #5 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #2 - #4 + #6 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #1 - #3 } }
      { \fp_to_dim:n { #2 - #4 } }
  }
\cs_new:Npn \__draw_path_ellipse_arciii:nnnnnn #1#2#3#4#5#6
  {
    \__draw_path_curveto:nnnnnn
      { \fp_to_dim:n { #1 - #3 - #5 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #2 - #4 - #6 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #1 - #3 * \c__draw_path_ellipse_fp - #5 } }
      { \fp_to_dim:n { #2 - #4 * \c__draw_path_ellipse_fp - #6 } }
      { \fp_to_dim:n { #1 - #5 } }
      { \fp_to_dim:n { #2 - #6 } }
  }
\cs_new:Npn \__draw_path_ellipse_arciv:nnnnnn #1#2#3#4#5#6
  {
    \__draw_path_curveto:nnnnnn
      { \fp_to_dim:n { #1 + #3 * \c__draw_path_ellipse_fp - #5 } }
      { \fp_to_dim:n { #2 + #4 * \c__draw_path_ellipse_fp - #6 } }
      { \fp_to_dim:n { #1 + #3 - #5 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #2 + #4 - #6 * \c__draw_path_ellipse_fp } }
      { \fp_to_dim:n { #1 + #3 } }
      { \fp_to_dim:n { #2 + #4 } }
  }
\fp_const:Nn \c__draw_path_ellipse_fp { \fp_use:c { c__draw_path_arc_90_fp } }
```

(*End definition for* `\draw_path_ellipse:nnn` *and others. This function is documented on page* **??**.)

`\draw_path_circle:nn` A shortcut.

```
\cs_new_protected:Npn \draw_path_circle:nn #1#2
  { \draw_path_ellipse:nnn {#1} { #2 , 0pt } { 0pt , #2 } }
```

(*End definition for* `\draw_path_circle:nn`. *This function is documented on page* 5.)

2.5 Rectangles

\draw_path_rectangle:nn
__draw_path_rectangle:nnnn
__draw_path_rectangle_rounded:nnnn

Building a rectangle can be a single operation, or for rounded versions will involve step-by-step construction.

```
\cs_new_protected:Npn \draw_path_rectangle:nn #1#2
  {
    \__draw_point_process:nnn
      {
        \bool_if:NTF \l__draw_corner_arc_bool
          { \__draw_path_rectangle_rounded:nnnn }
          { \__draw_path_rectangle:nnnn }
      }
      { \draw_point_transform:n {#1} }
      {#2}
  }
\cs_new_protected:Npn \__draw_path_rectangle:nnnn #1#2#3#4
  {
    \__draw_path_update_limits:nn {#1} {#2}
    \__draw_path_update_limits:nn { #1 + #3 } { #2 + #4 }
    \__draw_softpath_rectangle:nnnn {#1} {#2} {#3} {#4}
    \__draw_path_update_last:nn {#1} {#2}
  }
\cs_new_protected:Npn \__draw_path_rectangle_rounded:nnnn #1#2#3#4
  {
    \draw_path_moveto:n { #1 + #3 , #2 + #4 }
    \draw_path_lineto:n { #1 , #2 + #4 }
    \draw_path_lineto:n { #1 , #2 }
    \draw_path_lineto:n { #1 + #3 , #2 }
    \draw_path_close:
    \draw_path_moveto:n { #1 , #2 }
  }
```

(*End definition for* `\draw_path_rectangle:nn`, `\__draw_path_rectangle:nnnn`, *and* `\__draw_path_rectangle_rounded:nnnn`. *This function is documented on page* 6.)

\draw_path_rectangle_corners:nn
__draw_path_rectangle_corners:nnnn

Another shortcut wrapper.

```
\cs_new_protected:Npn \draw_path_rectangle_corners:nn #1#2
  {
    \__draw_point_process:nnn
      { \__draw_path_rectangle_corners:nnnnn {#1} }
      {#1} {#2}
  }
\cs_new_protected:Npn \__draw_path_rectangle_corners:nnnnn #1#2#3#4#5
  { \draw_path_rectangle:nn {#1} { #4 - #2 , #5 - #3 } }
```

(*End definition for* `\draw_path_rectangle_corners:nn` *and* `\__draw_path_rectangle_corners:nnnn`. *This function is documented on page* 6.)

2.6 Grids

\draw_path_grid:nnnn
__draw_path_grid:nnnnnn

A simple set of loops.

```
\cs_new_protected:Npn \draw_path_grid:nnnn #1#2#3#4
  {
    \__draw_point_process:nnn
```

```
      { \__draw_path_grid:nnnnnn {#1} {#2} }
      {#3} {#4}
  }
\cs_new_protected:Npn \__draw_path_grid:nnnnnn #1#2#3#4#5#6
  {
    \dim_step_inline:nnnn
      {#3} { \dim_compare:nNnF {#3} < {#5} { - } \dim_abs:n {#1} } {#5}
      {
        \draw_path_moveto:n { ##1 , #4 }
        \draw_path_lineto:n { ##1 , #6 }
      }
    \dim_step_inline:nnnn
      {#4} { \dim_compare:nNnF {#4} < {#6} { - } \dim_abs:n {#2} } {#6}
      {
        \draw_path_moveto:n { #3 , ##1 }
        \draw_path_lineto:n { #5 , ##1 }
      }
  }
```

(*End definition for* `\draw_path_grid:nnnn` *and* `\__draw_path_grid:nnnnnn`. *This function is documented on page* 6.)

2.7 Using paths

`\l__draw_path_use_clip_bool`
`\l__draw_path_use_fill_bool`
`\l__draw_path_use_stroke_bool`

Actions to pass to the driver.

```
\bool_new:N \l__draw_path_use_clip_bool
\bool_new:N \l__draw_path_use_fill_bool
\bool_new:N \l__draw_path_use_stroke_bool
```

(*End definition for* `\l__draw_path_use_clip_bool`, `\l__draw_path_use_fill_bool`, *and* `\l__draw_path_use_stroke_bool`.)

`\l__draw_path_use_bb_bool`
`\l__draw_path_use_clear_bool`

Actions handled at the macro layer.

```
\bool_new:N \l__draw_path_use_bb_bool
\bool_new:N \l__draw_path_use_clear_bool
```

(*End definition for* `\l__draw_path_use_bb_bool` *and* `\l__draw_path_use_clear_bool`.)

`\draw_path_use:n`
`\draw_path_use_clear:n`
`\__draw_path_use:n`
`\__draw_path_use_action_draw:`
`\__draw_path_use_stroke_bb:`
`\__draw_path_use_stroke_bb_aux:NnN`

There are a range of actions which can apply to a path: they are handled in a single function which can carry out several of them. The first step is to deal with the special case of clearing the path.

```
\cs_new_protected:Npn \draw_path_use:n #1
  {
    \tl_if_blank:nF {#1}
      { \__draw_path_use:n {#1} }
  }
\cs_new_protected:Npn \draw_path_use_clear:n #1
  {
    \bool_lazy_or:nnTF
      { \tl_if_blank_p:n {#1} }
      { \str_if_eq_p:nn {#1} { clear } }
      {
        \__draw_softpath_clear:
        \__draw_path_reset_limits:
```

```
      }
      { \__draw_path_use:n { #1 , clear } }
  }
```

Map over the actions and set up the data: mainly just booleans, but with the possibility to cover more complex cases. The business end of the function is a series of checks on the various flags, then taking the appropriate action(s).

```
\cs_new_protected:Npn \__draw_path_use:n #1
  {
    \bool_set_false:N \l__draw_path_use_clip_bool
    \bool_set_false:N \l__draw_path_use_fill_bool
    \bool_set_false:N \l__draw_path_use_stroke_bool
    \clist_map_inline:nn {#1}
      {
        \cs_if_exist:cTF { l__draw_path_use_ ##1 _ bool }
          { \bool_set_true:c { l__draw_path_use_ ##1 _ bool } }
          {
            \cs_if_exist_use:cF { __draw_path_use_action_ ##1 : }
              { \ERROR }
          }
      }
    \bool_lazy_and:nnT
      { \l__draw_update_bb_bool }
      { \l__draw_path_use_stroke_bool }
      { \__draw_path_use_stroke_bb: }
    \bool_if:NTF \l__draw_path_use_clear_bool
      { \__draw_softpath_use_clear: }
      { \__draw_softpath_use: }
    \bool_if:NT \l__draw_path_use_clip_bool
      { \driver_draw_clip: }
    \bool_lazy_or:nnT
      { \l__draw_path_use_fill_bool }
      { \l__draw_path_use_stroke_bool }
      {
        \use:c
          {
            driver_draw_
            \bool_if:NT \l__draw_path_use_fill_bool { fill }
            \bool_if:NT \l__draw_path_use_stroke_bool { stroke }
            :
          }
      }
  }
\cs_new_protected:Npn \__draw_path_use_action_draw:
  {
    \bool_set_true:N \l__draw_path_use_stroke_bool
  }
```

Where the path is relevant to size and is stroked, we need to allow for the part which overlaps the edge of the bounding box.

```
\cs_new_protected:Npn \__draw_path_use_stroke_bb:
  {
    \__draw_path_use_stroke_bb_aux:NnN x { max } +
    \__draw_path_use_stroke_bb_aux:NnN y { max } +
    \__draw_path_use_stroke_bb_aux:NnN x { min } -
```

```
    \__draw_path_use_stroke_bb_aux:NnN y { min } -
  }
\cs_new_protected:Npn \__draw_path_use_stroke_bb_aux:NnN #1#2#3
  {
    \dim_compare:nNnF { \dim_use:c { g__draw_ #1#2 _dim } } = { #3 -\c_max_dim }
      {
        \dim_gset:cn { g__draw_ #1#2 _dim }
          {
            \use:c { dim_ #2 :nn }
              { \dim_use:c { g__draw_ #1#2 _dim } }
              {
                  \dim_use:c { g__draw_path_ #1#2 _dim }
                #3 0.5 \g__draw_linewidth_dim
              }
          }
      }
  }
```

(*End definition for* `\draw_path_use:n` *and others. These functions are documented on page 6.*)

⟨/initex | package⟩

3 l3draw-points implementation

⟨*initex | package⟩

⟨@@=draw⟩

This sub-module covers more-or-less the same ideas as `pgfcorepoints.code.tex`, though the approach taken to returning values is different: point expressions here are processed by expansion and return a co-ordinate pair in the form `{`⟨*x*⟩`}{`⟨*y*⟩`}`. Equivalents of following pgf functions are deliberately omitted:

- `\pgfpointorigin`: Can be given explicitly as `{0pt}{0pt}`.
- `\pgfextractx`, `\pgfextracty`: Available by applying `\use_i:nn`/`\use_ii:nn` or similar to the `x`-type expansion of a point expression.
- `\pgfgetlastxy`: Unused in the entire pgf core, may be emulated by `x`-type expansion of a point expression, then using the result.

In addition, equivalents of the following *may* be added in future but are currently absent:

- `\pgfpointcylindrical`, `\pgfpointspherical`: The usefulness of these commands is not currently clear.
- `\pgfpointborderrectangle`, `\pgfpointborderellipse`: To be revisited once the semantics and use cases are clear.
- `\pgfqpoint`, `\pgfqpointscale`, `\pgfqpointpolar`, `\pgfqpointxy`, `\pgfqpointxyz`: The expandable approach taken in the code here, along with the absolute requirement for ε-TEX, means it is likely many use cases for these commands may be covered in other ways. This may be revisited as higher-level structures are constructed.

3.1 Support functions

__draw_point_process:nn
__draw_point_process_auxi:nn
__draw_point_process_auxi:fn
__draw_point_process_auxii:nw
__draw_point_process:nnn
__draw_point_process_auxiii:nnn
__draw_point_process_auxiii:ffn
__draw_point_process_auxiv:nw

Execute whatever code is passed to extract the x and y co-ordinates. The first argument here should itself absorb two arguments. There is also a version to deal with two co-ordinates: common enough to justify a separate function.

```
\cs_new:Npn \__draw_point_process:nn #1#2
  {
    \__draw_point_process_auxi:fn
      { \__draw_point_to_dim:n {#2} }
      {#1}
  }
\cs_new:Npn \__draw_point_process_auxi:nn #1#2
  { \__draw_point_process_auxii:nw {#2} #1 \q_stop }
\cs_generate_variant:Nn \__draw_point_process_auxi:nn { f }
\cs_new:Npn \__draw_point_process_auxii:nw #1 #2 , #3 \q_stop
  { #1 {#2} {#3} }
\cs_new:Npn \__draw_point_process:nnn #1#2#3
  {
    \__draw_point_process_auxiii:ffn
      { \__draw_point_to_dim:n {#2} }
      { \__draw_point_to_dim:n {#3} }
      {#1}
  }
\cs_new:Npn \__draw_point_process_auxiii:nnn #1#2#3
  { \__draw_point_process_auxiv:nw {#3} #1 \q_mark #2 \q_stop }
\cs_generate_variant:Nn \__draw_point_process_auxiii:nnn { ff }
\cs_new:Npn \__draw_point_process_auxiv:nw #1 #2 , #3 \q_mark #4 , #5 \q_stop
  { #1 {#2} {#3} {#4} {#5} }
```

(*End definition for* `\__draw_point_process:nn` *and others.*)

__draw_point_to_dim:n
__draw_point_to_dim_aux:n
__draw_point_to_dim_aux:f
__draw_point_to_dim_aux:w

Co-ordinates are always returned as two dimensions.

```
\cs_new:Npn \__draw_point_to_dim:n #1
  { \__draw_point_to_dim_aux:f { \fp_eval:n {#1} } }
\cs_new:Npn \__draw_point_to_dim_aux:n #1
  { \__draw_point_to_dim_aux:w #1 }
\cs_generate_variant:Nn \__draw_point_to_dim_aux:n { f }
\cs_new:Npn \__draw_point_to_dim_aux:w ( #1 , ~ #2 ) { #1pt , #2pt }
```

3.2 Co-ordinates

The most basic way of giving points is as simple (x, y) co-ordinates.

Simply turn the given values into dimensions.

\draw_point:nn

```
\cs_new:Npn \draw_point:nn #1#2
  { \__draw_point_to_dim:n { #1 , #2 } }
```

3.3 Polar co-ordinates

Polar co-ordinates may have either one or two lengths, so there is a need to do a simple split before the calculation. As the angle gets used twice, save on any expression evaluation there and force expansion.

\draw_point_polar:nn
\draw_point_polar:nnn
__draw_draw_polar:nnn
__draw_draw_polar:fnn

```
\cs_new:Npn \draw_point_polar:nn #1#2
```

```
  { \draw_point_polar:nnn {#1} {#2} {#2} }
\cs_new:Npn \draw_point_polar:nnn #1#2#3
  { \__draw_draw_polar:fnn { \fp_eval:n {#1} } {#2} {#3} }
\cs_new:Npn \__draw_draw_polar:nnn #1#2#3
  { \__draw_point_to_dim:n { cosd(#1) * (#2) , sind(#1) * (#3) } }
\cs_generate_variant:Nn \__draw_draw_polar:nnn { f }
```

3.4 Point expression arithmetic

These functions all take point expressions as arguments.

Simple mathematics.

\draw_point_add:nn
\draw_point_diff:nn
\draw_point_scale:nn

```
\cs_new:Npn \draw_point_add:nn #1#2
  { \__draw_point_to_dim:n { (#1) + (#2) } }
\cs_new:Npn \draw_point_diff:nn #1#2
  { \__draw_point_to_dim:n { (#2) - (#1) } }
\cs_new:Npn \draw_point_scale:nn #1#2
  { \__draw_point_to_dim:n { #1 * (#2) } }
```

\draw_point_unit_vector:n
__draw_point_unit_vector:nn

Only a single point expression so the expansion is done here. The outcome is the normalised vector from $(0,0)$ in the direction of the point, *i.e.*

$$P_x = \frac{x}{\sqrt{x^2+y^2}} \quad P_y = \frac{y}{\sqrt{x^2+y^2}}$$

```
\cs_new:Npn \draw_point_unit_vector:n #1
  { \__draw_point_process:nn { \__draw_point_unit_vector:nn } {#1} }
\cs_new:Npn \__draw_point_unit_vector:nn #1#2
  {
    \__draw_point_to_dim:n
      { ( #1 , #2 ) / (sqrt(#1 * #1 + #2 * #2)) }
  }
```

3.5 Intersection calculations

\draw_point_intersect_lines:nnnn
__draw_point_intersect_lines:nnnnnn
__draw_point_intersect_lines:nnnnnnnn
__draw_point_intersect_lines_aux:nnnnnn
__draw_point_intersect_lines_aux:ffffff

The intersection point P between a line joining points (x_1, y_1) and (x_2, y_2) with a second line joining points (x_3, y_3) and (x_4, y_4) can be calculated using the formulae

$$P_x = \frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_3y_4 - y_3x_4)(x_1 - x_2)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

and

$$P_y = \frac{(x_1y_2 - y_1x_2)(y_3 - y_5) - (x_3y_4 - y_3x_4)(y_1 - y_2)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

The work therefore comes down to expanding the incoming data, then pre-calculating as many parts as possible before the final work to find the intersection. (Expansion and argument re-ordering is much less work than additional floating point calculations.)

```
\cs_new:Npn \draw_point_intersect_lines:nnnn #1#2#3#4
  {
    \__draw_point_process:nnn
      {
        \__draw_point_process:nnn
          { \__draw_point_intersect_lines:nnnnnnnn } {#3} {#4}
```

```
      }
      {#1} {#2}
  }
```

At this stage we have all of the information we need, fully expanded:

#1 x_3

#2 y_3

#3 x_4

#4 y_4

#5 x_1

#6 y_1

#7 x_2

#8 y_2

so now just have to do all of the calculation.

```
\cs_new:Npn \__draw_point_intersect_lines:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_lines_aux:ffffff
      { \fp_eval:n { #1 * #4 - #2 * #3 } }
      { \fp_eval:n { #5 * #8 - #6 * #7 } }
      { \fp_eval:n { #1 - #3 } }
      { \fp_eval:n { #5 - #7 } }
      { \fp_eval:n { #2 - #4 } }
      { \fp_eval:n { #6 - #8 } }
  }
\cs_new:Npn \__draw_point_intersect_lines_aux:nnnnnn #1#2#3#4#5#6
  {
    \__draw_point_to_dim:n
      {
        ( #2 * #3 - #1 * #4 , #2 * #5 - #1 * #6 )
          / ( #4 * #5 - #6 * #3 )
      }
  }
\cs_generate_variant:Nn \__draw_point_intersect_lines_aux:nnnnnn { ffffff }
```

\draw_point_intersect_circles:nnnnn
__draw_point_intersect_circles_auxi:nnnnnnn
__draw_point_intersect_circles_auxii:nnnnnnn
__draw_point_intersect_circles_auxii:ffnnnnn
__draw_point_intersect_circles_auxiii:nnnnnnn
__draw_point_intersect_circles_auxiii:ffnnnnn
__draw_point_intersect_circles_auxiv:nnnnnnnn
__draw_point_intersect_circles_auxiv:fnnnnnnn
__draw_point_intersect_circles_auxv:nnnnnnnnn
__draw_point_intersect_circles_auxv:ffnnnnnnn
__draw_point_intersect_circles_auxvi:nnnnnnnn
__draw_point_intersect_circles_auxvi:fnnnnnnn
__draw_point_intersect_circles_auxvii:nnnnnnn
__draw_point_intersect_circles_auxvii:fffnnnn

Another long expansion chain to get the values in the right places. We have two circles, the first with center (a, b) and radius r, the second with center (c, d) and radius s. We use the intermediate values

$$e = c - a$$
$$f = d - b$$
$$p = \sqrt{e^2 + f^2}$$
$$k = \frac{p^2 + r^2 - s^2}{2p}$$

in either

$$P_x = a + \frac{ek}{p} + \frac{f}{p}\sqrt{r^2 - k^2}$$
$$P_y = b + \frac{fk}{p} - \frac{e}{p}\sqrt{r^2 - k^2}$$

or

$$P_x = a + \frac{ek}{p} - \frac{f}{p}\sqrt{r^2 - k^2}$$
$$P_y = b + \frac{fk}{p} + \frac{e}{p}\sqrt{r^2 - k^2}$$

depending on which solution is required. The rest of the work is simply forcing the appropriate expansion and shuffling arguments.

```
\cs_new:Npn \draw_point_intersect_circles:nnnnn #1#2#3#4#5
  {
    \__draw_point_process:nnn
      { \__draw_point_intersect_circles_auxi:nnnnnnn {#2} {#4} {#5} }
      {#1} {#3}
  }
\cs_new:Npn \__draw_point_intersect_circles_auxi:nnnnnnn #1#2#3#4#5#6#7
  {
    \__draw_point_intersect_circles_auxii:ffnnnnn
      { \fp_eval:n {#1} } { \fp_eval:n {#2} } {#4} {#5} {#6} {#7} {#3}
  }
```

At this stage we have all of the information we need, fully expanded:

#1 r

#2 s

#3 a

#4 b

#5 c

#6 d

#7 n

Once we evaluate e and f, the co-ordinate (c, d) is no longer required: handy as we will need various intermediate values in the following.

```
\cs_new:Npn \__draw_point_intersect_circles_auxii:nnnnnnn #1#2#3#4#5#6#7
  {
    \__draw_point_intersect_circles_auxiii:ffnnnnn
      { \fp_eval:n { #5 - #3 } }
      { \fp_eval:n { #6 - #4 } }
      {#1} {#2} {#3} {#4} {#7}
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxii:nnnnnnn { ff }
\cs_new:Npn \__draw_point_intersect_circles_auxiii:nnnnnnn #1#2#3#4#5#6#7
  {
```

```
    \__draw_point_intersect_circles_auxiv:fnnnnnnn
      { \fp_eval:n { sqrt( #1 * #1 + #2 * #2 ) } }
      {#1} {#2} {#3} {#4} {#5} {#6} {#7}
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxiii:nnnnnnn { ff }
```

We now have p: we pre-calculate $1/p$ as it is needed a few times and is relatively expensive. We also need r^2 twice so deal with that here too.

```
\cs_new:Npn \__draw_point_intersect_circles_auxiv:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_circles_auxv:ffnnnnnnn
      { \fp_eval:n { 1 / #1 } }
      { \fp_eval:n { #4 * #4 } }
      {#1} {#2} {#3} {#5} {#6} {#7} {#8}
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxiv:nnnnnnnn { f }
\cs_new:Npn \__draw_point_intersect_circles_auxv:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_intersect_circles_auxvi:fnnnnnnn
      { \fp_eval:n { 0.5 * #1 * ( #2 + #3 * #3 - #6 * #6 ) } }
      {#1} {#2} {#4} {#5} {#7} {#8} {#9}
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxv:nnnnnnnnn { ff }
```

We now have all of the intermediate values we require, with one division carried out up-front to avoid doing this expensive step twice:

`#1` k

`#2` $1/p$

`#3` r^2

`#4` e

`#5` f

`#6` a

`#7` b

`#8` n

There are some final pre-calculations, k/p, $\frac{\sqrt{r^2-k^2}}{p}$ and the usage of n, then we can yield a result.

```
\cs_new:Npn \__draw_point_intersect_circles_auxvi:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_circles_auxvii:fffnnnn
      { \fp_eval:n { #1 * #2 } }
      { \int_if_odd:nTF {#8} { 1 } { -1 } }
      { \fp_eval:n { sqrt ( #3 - #1 * #1 ) * #2 } }
      {#4} {#5} {#6} {#7}
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxvi:nnnnnnnn { f }
\cs_new:Npn \__draw_point_intersect_circles_auxvii:nnnnnnn #1#2#3#4#5#6#7
  {
```

```
    \__draw_point_to_dim:n
      { #6 + #4 * #1 + #2 * #3 * #5 , #7 + #5 * #1 + -1 * #2 * #3 * #4 }
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxvii:nnnnnnn { fff }
```

3.6 Interpolation on a line (vector) or arc

Simple maths after expansion.

\draw_point_interpolate_line:nnn
__draw_point_interpolate_line_aux:nnnnn
__draw_point_interpolate_line_aux:fnnnn
__draw_point_interpolate_line_aux:nnnnnn
__draw_point_interpolate_line_aux:fnnnnn

```
\cs_new:Npn \draw_point_interpolate_line:nnn #1#2#3
  {
    \__draw_point_process:nnn
      { \__draw_point_interpolate_line_aux:fnnnn { \fp_eval:n {#1} } }
      {#2} {#3}
  }
\cs_new:Npn \__draw_point_interpolate_line_aux:nnnnn #1#2#3#4#5
  {
    \__draw_point_interpolate_line_aux:fnnnnn { \fp_eval:n { 1 - #1 } }
      {#1} {#2} {#3} {#4} {#5}
  }
\cs_generate_variant:Nn \__draw_point_interpolate_line_aux:nnnnn { f }
\cs_new:Npn \__draw_point_interpolate_line_aux:nnnnnn #1#2#3#4#5#6
  { \__draw_point_to_dim:n { #2 * #3 + #1 * #5 , #2 * #4 + #1 * #6 } }
\cs_generate_variant:Nn \__draw_point_interpolate_line_aux:nnnnnn { f }
```

Same idea but using the normalised length to obtain the scale factor.

\draw_point_interpolate_distance:nnn
__draw_point_interpolate_distance:nnnnn
_draw_point_interpolate_distance_aux:nnnnnnn
_draw_point_interpolate_distance_aux:nnnnnnn
__draw_point_interpolate_distance_aux:nnnnnn
__draw_point_interpolate_distance_aux:fnnnnn

```
\cs_new:Npn \draw_point_interpolate_distance:nnn #1#2#3
  {
    \__draw_point_process:nnn
      { \__draw_point_interpolate_distance:nnnnn {#1} }
      {#2} {#3}
  }
\cs_new:Npn \__draw_point_interpolate_distance:nnnnn #1#2#3#4#5
  {
    \__draw_point_interpolate_distance_aux:nnnnnnn
      { \fp_eval:n { #4 - #2 } }
      { \fp_eval:n { #5 - #3 } }
      {#2} {#3} {#4} {#5} {#1}
  }
\cs_new:Npn \__draw_point_interpolate_distance_aux:nnnnnnn #1#2#3#4#5#6#7
  {
    \__draw_point_interpolate_distance_aux:fnnnn
      { \fp_eval:n { (#7) / (sqrt ( #1 * #1 + #2 * #2 )) } }
      {#3} {#4} {#5} {#6}
  }
\cs_generate_variant:Nn \__draw_point_interpolate_distance_aux:nnnnnnn { ff }
\cs_new:Npn \__draw_point_interpolate_distance_aux:nnnnn #1#2#3#4#5
  { \__draw_point_to_dim:n { #2 + #1 * #4 , #3 + #1 * #5 } }
\cs_generate_variant:Nn \__draw_point_interpolate_distance_aux:nnnnn { f }
```

Finding a point on an ellipse arc is relatively easy: find the correct angle between the two given, use the sine and cosine of that angle, apply to the axes. We just have to work a bit with the co-ordinate expansion.

\draw_point_interpolate_arcaxes:nnnnnn
raw_point_interpolate_arcaxes_auxi:nnnnnnnnn
aw_point_interpolate_arcaxes_auxii:nnnnnnnnn
aw_point_interpolate_arcaxes_auxii:fnnnnnnnn
raw_point_interpolate_arcaxes_auxiii:nnnnnnn
raw_point_interpolate_arcaxes_auxiii:fnnnnnn
raw_point_interpolate_arcaxes_auxiv:nnnnnnnn
raw_point_interpolate_arcaxes_auxiv:ffnnnnnn

```
\cs_new:Npn \draw_point_interpolate_arcaxes:nnnnnn #1#2#3#4#5#6
```

```
  {
    \__draw_point_process:nnn
      {
        \__draw_point_process:nn
          { \__draw_point_interpolate_arcaxes_auxi:nnnnnnnnn {#1} {#5} {#6} }
          {#4}
      }
      {#2} {#3}
  }
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxi:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_interpolate_arcaxes_auxii:fnnnnnnnn
      { \fp_eval:n {#1} } {#2} {#3} {#6} {#7} {#8} {#9} {#4} {#5}
  }
```

At this stage, the three co-ordinate pairs are fully expanded but somewhat re-ordered:

#1 p

#2 θ_1

#3 θ_2

#4 x_c

#5 y_c

#6 x_{a1}

#7 y_{a1}

#8 x_{a2}

#9 y_{a2}

We are now in a position to find the target angle, and from that the sine and cosine required.

```
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxii:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_interpolate_arcaxes_auxiii:fnnnnnn
      { \fp_eval:n { #1 * (#3) + ( 1 - #1 ) * (#2) } }
      {#4} {#5} {#6} {#7} {#8} {#9}
  }
\cs_generate_variant:Nn \__draw_point_interpolate_arcaxes_auxii:nnnnnnnnn { f }
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxiii:nnnnnnn #1#2#3#4#5#6#7
  {
    \__draw_point_interpolate_arcaxes_auxiv:ffnnnnnn
      { \fp_eval:n { cosd (#1) } }
      { \fp_eval:n { sind (#1) } }
      {#2} {#3} {#4} {#5} {#6} {#7}
  }
\cs_generate_variant:Nn \__draw_point_interpolate_arcaxes_auxiii:nnnnnnn { f }
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxiv:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_to_dim:n
      { #3 + #1 * #5 + #2 * #7 , #4 + #1 * #6 + #2 * #8 }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_arcaxes_auxiv:nnnnnnnn { ff }
```

(*End definition for* `\__draw_point_to_dim:n` *and others. These functions are documented on page 3.*)

\draw_point_interpolate_curve:nnnnn
\draw_point_interpolate_curve_auxi:nnnnnnnnn
draw_point_interpolate_curve_auxii:nnnnnnnnn
draw_point_interpolate_curve_auxii:fnnnnnnnn
\draw_point_interpolate_curve_auxiii:nnnnnn
\draw_point_interpolate_curve_auxiii:fnnnnn
\draw_point_interpolate_curve_auxiv:nnnnnn
\draw_point_interpolate_curve_auxv:nnw
\draw_point_interpolate_curve_auxv:ffw
\draw_point_interpolate_curve_auxvi:n
draw_point_interpolate_curve_auxvii:nnnnnnnn
\draw_point_interpolate_curve_auxviii:nnnnnn
\draw_point_interpolate_curve_auxviii:ffnnnn

Here we start with a proportion of the curve (p) and four points

1. The initial point (x_1, y_1)
2. The first control point (x_2, y_2)
3. The second control point (x_3, y_3)
4. The final point (x_4, y_4)

The first phase is to expand out all of these values.

```
\cs_new:Npn \draw_point_interpolate_curve:nnnnnn #1#2#3#4#5
  {
    \__draw_point_process:nnn
      {
        \__draw_point_process:nnn
          { \__draw_point_interpolate_curve_auxi:nnnnnnnnn {#1} }
          {#4} {#5}
      }
      {#2} {#3}
  }
\cs_new:Npn \__draw_point_interpolate_curve_auxi:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_interpolate_curve_auxii:fnnnnnnnn
      { \fp_eval:n {#1} }
      {#6} {#7} {#8} {#9} {#2} {#3} {#4} {#5}
  }
```

At this stage, everything is fully expanded and back in the input order. The approach to finding the required point is iterative. We carry out three phases. In phase one, we need all of the input co-ordinates

$$\begin{aligned}
x'_1 &= (1-p)x_1 + px_2 \\
y'_1 &= (1-p)y_1 + py_2 \\
x'_2 &= (1-p)x_2 + px_3 \\
y'_2 &= (1-p)y_2 + py_3 \\
x'_3 &= (1-p)x_3 + px_4 \\
y'_3 &= (1-p)y_3 + py_4
\end{aligned}$$

In the second stage, we can drop the final point

$$\begin{aligned}
x''_1 &= (1-p)x'_1 + px'_2 \\
y''_1 &= (1-p)y'_1 + py'_2 \\
x''_2 &= (1-p)x'_2 + px'_3 \\
y''_2 &= (1-p)y'_2 + py'_3
\end{aligned}$$

and for the final stage only need one set of calculations

$$\begin{aligned}
P_x &= (1-p)x''_1 + px''_2 \\
P_y &= (1-p)y''_1 + py''_2
\end{aligned}$$

Of course, this does mean a lot of calculations and expansion!

```
\cs_new:Npn \__draw_point_interpolate_curve_auxii:nnnnnnnnn
  #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_interpolate_curve_auxiii:fnnnnn
      { \fp_eval:n { 1 - #1 } }
      {#1}
      { {#2} {#3} } { {#4} {#5} } { {#6} {#7} } { {#8} {#9} }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_curve_auxii:nnnnnnnnn { f }
%    \begin{macrocode}
%   We need to do the first cycle, but haven't got enough arguments to keep
%   everything in play at once. So her ewe use a but of argument re-ordering
%   and a single auxiliary to get the job done.
%    \begin{macrocode}
\cs_new:Npn \__draw_point_interpolate_curve_auxiii:nnnnnn #1#2#3#4#5#6
  {
    \__draw_point_interpolate_curve_auxiv:nnnnnn {#1} {#2} #3 #4
    \__draw_point_interpolate_curve_auxiv:nnnnnn {#1} {#2} #4 #5
    \__draw_point_interpolate_curve_auxiv:nnnnnn {#1} {#2} #5 #6
    \prg_do_nothing:
    \__draw_point_interpolate_curve_auxvi:n { {#1} {#2} }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_curve_auxiii:nnnnnn { f }
\cs_new:Npn \__draw_point_interpolate_curve_auxiv:nnnnnn #1#2#3#4#5#6
  {
    \__draw_point_interpolate_curve_auxv:ffw
      { \fp_eval:n { #1 * #3 + #2 * #5 } }
      { \fp_eval:n { #1 * #4 + #2 * #6 } }
  }
\cs_new:Npn \__draw_point_interpolate_curve_auxv:nnw
  #1#2#3 \prg_do_nothing: #4#5
  {
    #3
    \prg_do_nothing:
    #4 { #5 {#1} {#2} }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_curve_auxv:nnw { ff }
%    \begin{macrocode}
%   Get the arguments back into the right places and to the second and
%   third cycles directly.
%    \begin{macrocode}
\cs_new:Npn \__draw_point_interpolate_curve_auxvi:n #1
  { \__draw_point_interpolate_curve_auxvii:nnnnnnnn #1 }
\cs_new:Npn \__draw_point_interpolate_curve_auxvii:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_interpolate_curve_auxviii:ffffnn
      { \fp_eval:n { #1 * #5 + #2 * #3 } }
      { \fp_eval:n { #1 * #6 + #2 * #4 } }
      { \fp_eval:n { #1 * #7 + #2 * #5 } }
      { \fp_eval:n { #1 * #8 + #2 * #6 } }
      {#1} {#2}
  }
\cs_new:Npn \__draw_point_interpolate_curve_auxviii:nnnnnn #1#2#3#4#5#6
```

```
  {
    \__draw_point_to_dim:n
      { #5 * #3 + #6 * #1 , #5 * #4 + #6 * #2 }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_curve_auxviii:nnnnnn { ffff }
```

(*End definition for* `\draw_point_interpolate_curve:nnnnn` *and others. These functions are documented on page* **??**.)

3.7 Vector support

As well as co-ordinates relative to the drawing

`\l__draw_xvec_x_dim` Base vectors to map to the underlying two-dimensional drawing space.

```
\dim_new:N \l__draw_xvec_x_dim
\dim_new:N \l__draw_xvec_y_dim
\dim_new:N \l__draw_yvec_x_dim
\dim_new:N \l__draw_yvec_y_dim
\dim_new:N \l__draw_zvec_x_dim
\dim_new:N \l__draw_zvec_y_dim
```

(*End definition for* `\l__draw_xvec_x_dim` *and others.*)

`\draw_xvec:n` Calculate the underlying position and store it.

```
\cs_new_protected:Npn \draw_xvec:n #1
  { \__draw_vec:nn { x } {#1} }
\cs_new_protected:Npn \draw_yvec:n #1
  { \__draw_vec:nn { y } {#1} }
\cs_new_protected:Npn \draw_zvec:n #1
  { \__draw_vec:nn { z } {#1} }
\cs_new_protected:Npn \__draw_vec:nn #1#2
  {
    \__draw_point_process:nn { \__draw_vec:nnn {#1} } {#2}
  }
\cs_new_protected:Npn \__draw_vec:nnn #1#2#3
  {
    \dim_set:cn { l__draw_ #1 vec_x_dim } {#2}
    \dim_set:cn { l__draw_ #1 vec_y_dim } {#3}
  }
```

(*End definition for* `\draw_xvec:n` *and others. These functions are documented on page* **??**.)

Initialise the vectors.

```
\draw_xvec:n { 1cm , 0cm }
\draw_yvec:n { 0cm , 1cm }
\draw_zvec:n { -0.385cm , -0.385cm }
```

`\draw_point_vec:nn` Force a single evaluation of each factor, then use these to work out the underlying point.

```
\cs_new:Npn \draw_point_vec:nn #1#2
  { \__draw_point_vec:ff { \fp_eval:n {#1} } { \fp_eval:n {#2} } }
\cs_new:Npn \__draw_point_vec:nn #1#2
  {
    \__draw_point_to_dim:n
      {
        #1 * \l__draw_xvec_x_dim + #2 * \l__draw_yvec_x_dim ,
```

```
        #1 * \l__draw_xvec_y_dim + #2 * \l__draw_yvec_y_dim
      }
  }
\cs_generate_variant:Nn \__draw_point_vec:nn { ff }
\cs_new:Npn \draw_point_vec:nnn #1#2#3
  {
    \__draw_point_vec:fff
      { \fp_eval:n {#1} } { \fp_eval:n {#2} } { \fp_eval:n {#3} }
  }
\cs_new:Npn \__draw_point_vec:nnn #1#2#3
  {
    \__draw_point_to_dim:n
      {
            #1 * \l__draw_xvec_x_dim
          + #2 * \l__draw_yvec_x_dim
          + #3 * \l__draw_zvec_x_dim
        ,
            #1 * \l__draw_xvec_y_dim
          + #2 * \l__draw_yvec_y_dim
          + #3 * \l__draw_zvec_y_dim
    }
  }
\cs_generate_variant:Nn \__draw_point_vec:nnn { fff }
```

(*End definition for* `\draw_point_vec:nn` *and others. These functions are documented on page 3.*)

`\draw_point_vec_polar:nn`
`\draw_point_vec_polar:nnn`
`\__draw_point_vec_polar:nnn`
`\__draw_point_vec_polar:fnn`

Much the same as the core polar approach.

```
\cs_new:Npn \draw_point_vec_polar:nn #1#2
  { \draw_point_vec_polar:nnn {#1} {#2} {#2} }
\cs_new:Npn \draw_point_vec_polar:nnn #1#2#3
  { \__draw_draw_vec_polar:fnn { \fp_eval:n {#1} } {#2} {#3} }
\cs_new:Npn \__draw_draw_vec_polar:nnn #1#2#3
  {
    \__draw_point_to_dim:n
      {
        cosd(#1) * (#2) * \l__draw_xvec_x_dim ,
        sind(#1) * (#3) * \l__draw_yvec_y_dim
     }
  }
\cs_generate_variant:Nn \__draw_draw_vec_polar:nnn { f }
```

(*End definition for* `\draw_point_vec_polar:nn`*,* `\draw_point_vec_polar:nnn`*, and* `\__draw_point_vec_polar:nnn`*. These functions are documented on page 4.*)

3.8 Transformations

`\draw_point_transform:n`
`\__draw_point_transform:nn`

Applies a transformation matrix to a point: see `l3draw-transforms` for the business end. Where possible, we avoid the relatively expensive multiplication step.

```
\cs_new:Npn \draw_point_transform:n #1
  {
    \__draw_point_process:nn
      { \__draw_point_transform:nn } {#1}
  }
\cs_new:Npn \__draw_point_transform:nn #1#2
```

```
  {
    \bool_if:NTF \l__draw_transformcm_active_bool
      {
        \__draw_point_to_dim:n
          {
            (
                \l__draw_transformcm_aa_fp * #1
              + \l__draw_transformcm_ba_fp * #2
              + \l__draw_transformcm_xshift_dim
            )
            ,
            (
                \l__draw_transformcm_ab_fp * #1
              + \l__draw_transformcm_bb_fp * #2
              + \l__draw_transformcm_yshift_dim
            )
        }
      }
      {
        \__draw_point_to_dim:n
          {
              (#1, #2)
            + ( \l__draw_transformcm_xshift_dim ,
                \l__draw_transformcm_yshift_dim )
          }
      }
  }
```

(*End definition for* `\draw_point_transform:n` *and* `\__draw_point_transform:nn`. *This function is documented on page* 3.)

`\__draw_point_transform_noshift:n`
`\__draw_point_transform_noshift:nn`

A version with no shift: used for internal purposes.

```
\cs_new:Npn \__draw_point_transform_noshift:n #1
  {
    \__draw_point_process:nn
      { \__draw_point_transform_noshift:nn } {#1}
  }
\cs_new:Npn \__draw_point_transform_noshift:nn #1#2
  {
    \bool_if:NTF \l__draw_transformcm_active_bool
      {
        \__draw_point_to_dim:n
          {
            (
                \l__draw_transformcm_aa_fp * #1
              + \l__draw_transformcm_ba_fp * #2
            )
            ,
            (
                \l__draw_transformcm_ab_fp * #1
              + \l__draw_transformcm_bb_fp * #2
            )
        }
      }
```

```
      { \__draw_point_to_dim:n { (#1, #2) } }
  }
```

(*End definition for* `\__draw_point_transform_noshift:n` *and* `\__draw_point_transform_noshift:nn`.)

⟨/initex | package⟩

4 l3draw-scopes implementation

⟨*initex | package⟩

⟨@@=draw⟩

4.1 Drawing environment

`\g__draw_xmax_dim`
`\g__draw_xmin_dim`
`\g__draw_ymax_dim`
`\g__draw_ymin_dim`

Used to track the overall (official) size of the image created: may not actually be the natural size of the content.

```
\dim_new:N \g__draw_xmax_dim
\dim_new:N \g__draw_xmin_dim
\dim_new:N \g__draw_ymax_dim
\dim_new:N \g__draw_ymin_dim
```

(*End definition for* `\g__draw_xmax_dim` *and others.*)

`\l__draw_update_bb_bool`

Flag to indicate that a path (or similar) should update the bounding box of the drawing.

```
\bool_new:N \l__draw_update_bb_bool
```

(*End definition for* `\l__draw_update_bb_bool`.)

`\l__draw_main_box`

Box for setting the drawing.

```
\box_new:N \l__draw_main_box
```

(*End definition for* `\l__draw_main_box`.)

`\draw_begin:`
`\draw_end:`

Drawings are created by setting them into a box, then adjusting the box before inserting into the surroundings. At present the content is simply collected then dumped: work will be required to manipulate the size as this data becomes more defined. It may be that a coffin construct is better here in the longer term: that may become clearer as the code is completed. Another obvious question is whether/where vertical mode should be ended (*i.e.* should this behave like a raw `\vbox` or like a coffin). In contrast to pgf, we use a vertical box here: material between explicit instructions should not be present anyway. (Consider adding an `\everypar` hook as done for the LaTeX 2_ε preamble.)

```
\cs_new_protected:Npn \draw_begin:
  {
    \vbox_set:Nw \l__draw_main_box
      \driver_draw_begin:
      \dim_gset:Nn \g__draw_xmax_dim { -\c_max_dim }
      \dim_gset:Nn \g__draw_xmin_dim {  \c_max_dim }
      \dim_gset:Nn \g__draw_ymax_dim { -\c_max_dim }
      \dim_gset:Nn \g__draw_ymin_dim {  \c_max_dim }
      \bool_set_true:N \l__draw_update_bb_bool
      \draw_transform_reset:
      \draw_linewidth:n { \l_draw_default_linewidth_dim }
  }
\cs_new_protected:Npn \draw_end:
```

```
{
    \driver_draw_end:
  \vbox_set_end:
  \hbox_set:Nn \l__draw_main_box
    {
      \skip_horizontal:n { -\g__draw_xmin_dim }
      \box_move_down:nn { \g__draw_ymin_dim }
        { \box_use_drop:N \l__draw_main_box }
    }
  \box_set_ht:Nn \l__draw_main_box
    { \g__draw_ymax_dim - \g__draw_ymin_dim }
  \box_set_dp:Nn \l__draw_main_box { 0pt }
  \box_set_wd:Nn \l__draw_main_box
    { \g__draw_xmax_dim - \g__draw_xmin_dim }
  \mode_leave_vertical:
  \box_use_drop:N \l__draw_main_box
}
```

(*End definition for* `\draw_begin:` *and* `\draw_end:`*. These functions are documented on page* 1.)

⟨/initex | package⟩

5 l3draw-softpath implementation

⟨*initex | package⟩

⟨@@=draw⟩

There are two linked aims in the code here. The most significant is to provide a way to modify paths, for example to shorten the ends or round the corners. This means that the path cannot be written piecemeal as specials, but rather needs to be held in macros. The second aspect that follows from this is performance: simply adding to a single macro a piece at a time will have poor performance as the list gets long. Paths need to be global (as specials are), so we cannot use l3tl-build or a similar approach. Instead, we use the same idea as pgf: use a series of buffer macros such that in most cases we don't add tokens to the main list. This will get slow only for *enormous* paths.

Each marker (operation) token takes two arguments, which makes processing more straight-forward. As such, some operations have dummy arguments, whilst others have to be split over several tokens. As the code here is at a low level, all dimension arguments are assumed to be explicit and fully-expanded.

`\g__draw_softpath_main_tl` The soft path itself.

`\g__draw_softpath_buffer_a_tl`
`\g__draw_softpath_buffer_b_tl`

```
\tl_new:N \g__draw_softpath_main_tl
\tl_new:N \g__draw_softpath_buffer_a_tl
\tl_new:N \g__draw_softpath_buffer_b_tl
```

(*End definition for* `\g__draw_softpath_main_tl`*,* `\g__draw_softpath_buffer_a_tl`*, and* `\g__draw_softpath_buffer_b_tl`.)

`\g__draw_softpath_buffer_a_int` Tracking data.

`\g__draw_softpath_buffer_b_int`

```
\int_new:N \g__draw_softpath_buffer_a_int
\int_new:N \g__draw_softpath_buffer_b_int
```

(*End definition for* `\g__draw_softpath_buffer_a_int` *and* `\g__draw_softpath_buffer_b_int`.)

__draw_softpath_add:n
__draw_softpath_add:x
__draw_softpath_concat:n
__draw_softpath_reset_buffers:

The softpath itself is quite simple. We use three token lists to hold the data: two buffers of limited length, and the main list of arbitrary size. Most of the time this will mean that we don't add to the full list, so performance will be acceptable.

```
\cs_new_protected:Npn \__draw_softpath_add:n #1
  {
    \int_compare:nNnTF \g__draw_softpath_buffer_a_int < { 40 }
      {
        \int_gincr:N \g__draw_softpath_buffer_a_int
        \tl_gput_right:Nn \g__draw_softpath_buffer_a_tl {#1}
      }
      {
        \int_compare:nNnTF \g__draw_softpath_buffer_b_int < { 40 }
          {
            \int_gincr:N \g__draw_softpath_buffer_b_int
            \tl_gset:Nx \g__draw_softpath_buffer_b_tl
              {
                \exp_not:V \g__draw_softpath_buffer_b_tl
                \exp_not:V \g__draw_softpath_buffer_a_tl
                \exp_not:n {#1}
              }
            \int_gzero:N \g__draw_softpath_buffer_a_int
            \tl_gclear:N \g__draw_softpath_buffer_a_tl
          }
          { \__draw_softpath_concat:n {#1} }
      }
  }
\cs_generate_variant:Nn \__draw_softpath_add:n { x }
\cs_new_protected:Npn \__draw_softpath_concat:n #1
  {
    \tl_gset:Nx \g__draw_softpath_main_tl
      {
        \exp_not:V \g__draw_softpath_main_tl
        \exp_not:V \g__draw_softpath_buffer_b_tl
        \exp_not:V \g__draw_softpath_buffer_a_tl
        \exp_not:n {#1}
      }
    \__draw_softpath_reset_buffers:
  }
\cs_new_protected:Npn \__draw_softpath_reset_buffers:
  {
    \int_gzero:N \g__draw_softpath_buffer_a_int
    \tl_gclear:N \g__draw_softpath_buffer_a_tl
    \int_gzero:N \g__draw_softpath_buffer_b_int
    \tl_gclear:N \g__draw_softpath_buffer_b_tl
  }
```

(*End definition for* `\__draw_softpath_add:n`, `\__draw_softpath_concat:n`, *and* `\__draw_softpath_reset_buffers:`.)

__draw_softpath_get:N
__draw_softpath_set_eq:N

Save and restore functions.

```
\cs_new_protected:Npn \__draw_softpath_get:N #1
  {
    \__draw_softpath_concat:n { }
    \tl_set_eq:NN #1 \g__draw_softpath_main_tl
```

```
  }
\cs_new_protected:Npn \__draw_softpath_set_eq:N #1
  {
    \tl_gset_eq:NN \g__draw_softpath_main_tl #1
    \__draw_softpath_reset_buffers:
  }
```

(*End definition for* `\__draw_softpath_get:N` *and* `\__draw_softpath_set_eq:N`.)

`\__draw_softpath_use:` `\__draw_softpath_clear:` `\__draw_softpath_use_clear:`

Using and clearing is trivial.

```
\cs_new_protected:Npn \__draw_softpath_use:
  {
    \g__draw_softpath_main_tl
    \g__draw_softpath_buffer_b_tl
    \g__draw_softpath_buffer_a_tl
  }
\cs_new_protected:Npn \__draw_softpath_clear:
  {
    \tl_gclear:N \g__draw_softpath_main_tl
    \tl_gclear:N \g__draw_softpath_buffer_a_tl
    \tl_gclear:N \g__draw_softpath_buffer_b_tl
  }
\cs_new_protected:Npn \__draw_softpath_use_clear:
  {
    \__draw_softpath_use:
    \__draw_softpath_clear:
  }
```

(*End definition for* `\__draw_softpath_use:`, `\__draw_softpath_clear:`, *and* `\__draw_softpath_use_clear:`.)

`\g__draw_softpath_lastx_dim` `\g__draw_softpath_lasty_dim`

For tracking the end of the path (to close it).

```
\dim_new:N \g__draw_softpath_lastx_dim
\dim_new:N \g__draw_softpath_lasty_dim
```

(*End definition for* `\g__draw_softpath_lastx_dim` *and* `\g__draw_softpath_lasty_dim`.)

`\g__draw_softpath_move_bool`

Track if moving a point should update the close position.

```
\bool_new:N \g__draw_softpath_move_bool
\bool_gset_true:N \g__draw_softpath_move_bool
```

(*End definition for* `\g__draw_softpath_move_bool`.)

`\__draw_softpath_curveto:nnnnnn` `\__draw_softpath_lineto:nn` `\__draw_softpath_moveto:nn` `\__draw_softpath_rectangle:nnnn` `\__draw_softpath_roundpoint:nn` `\__draw_softpath_roundpoint:VV`

The various parts of a path expressed as the appropriate soft path functions.

```
\cs_new_protected:Npn \__draw_softpath_closepath:
  {
    \__draw_softpath_add:x
      {
        \__draw_softpath_close_op:nn
          { \dim_use:N \g__draw_softpath_lastx_dim }
          { \dim_use:N \g__draw_softpath_lasty_dim }
      }
  }
\cs_new_protected:Npn \__draw_softpath_curveto:nnnnnn #1#2#3#4#5#6
  {
```

```
    \__draw_softpath_add:n
      {
        \__draw_softpath_curveto_opi:nn {#1} {#2}
        \__draw_softpath_curveto_opii:nn {#3} {#4}
        \__draw_softpath_curveto_opiii:nn {#5} {#6}
      }
  }
\cs_new_protected:Npn \__draw_softpath_lineto:nn #1#2
  {
    \__draw_softpath_add:n
      { \__draw_softpath_lineto_op:nn {#1} {#2} }
  }
\cs_new_protected:Npn \__draw_softpath_moveto:nn #1#2
  {
    \__draw_softpath_add:n
      { \__draw_softpath_moveto_op:nn {#1} {#2} }
    \bool_if:NT \g__draw_softpath_move_bool
      {
        \dim_gset:Nn \g__draw_softpath_lastx_dim {#1}
        \dim_gset:Nn \g__draw_softpath_lasty_dim {#2}
      }
  }
\cs_new_protected:Npn \__draw_softpath_rectangle:nnnn #1#2#3#4
  {
    \__draw_softpath_add:n
      {
        \__draw_softpath_rectangle_opi:nn {#1} {#2}
        \__draw_softpath_rectangle_opii:nn {#3} {#4}
      }
  }
\cs_new_protected:Npn \__draw_softpath_roundpoint:nn #1#2
  {
    \__draw_softpath_add:n
      { \__draw_softpath_roundpoint_op:nn {#1} {#2} }
  }
\cs_generate_variant:Nn \__draw_softpath_roundpoint:nn { VV }
```

(*End definition for* `\__draw_softpath_curveto:nnnnnn` *and others.*)

`\__draw_softpath_close_op:nn`
`\__draw_softpath_curveto_opi:nn`
`\__draw_softpath_curveto_opii:nn`
`\__draw_softpath_curveto_opiii:nn`
`\__draw_softpath_lineto_op:nn`
`\__draw_softpath_moveto_op:nn`
`\__draw_softpath_roundpoint_op:nn`
`\__draw_softpath_rectangle_opi:nn`
`\__draw_softpath_rectangle_opii:nn`
`\__draw_softpath_curveto_opi:nnNnnNnn`
`\__draw_softpath_rectangle_opi:nnNnn`

The markers for operations: all the top-level ones take two arguments.

```
\cs_new_protected:Npn \__draw_softpath_close_op:nn #1#2
  { \driver_draw_closepath: }
\cs_new_protected:Npn \__draw_softpath_curveto_opi:nn #1#2
  { \__draw_softpath_curveto_opi:nnNnnNnn {#1} {#2} }
\cs_new_protected:Npn \__draw_softpath_curveto_opi:nnNnnNnn #1#2#3#4#5#6#7#8
  { \driver_draw_curveto:nnnnnn {#1} {#2} {#4} {#5} {#7} {#8} }
\cs_new_protected:Npn \__draw_softpath_curveto_opii:nn #1#2 { }
\cs_new_protected:Npn \__draw_softpath_curveto_opiii:nn #1#2 { }
\cs_new_protected:Npn \__draw_softpath_lineto_op:nn #1#2
  { \driver_draw_lineto:nn {#1} {#2} }
\cs_new_protected:Npn \__draw_softpath_moveto_op:nn #1#2
  { \driver_draw_moveto:nn {#1} {#2} }
\cs_new_protected:Npn \__draw_softpath_roundpoint_op:nn #1#2 { }
\cs_new_protected:Npn \__draw_softpath_rectangle_opi:nn #1#2
```

```
  { \__draw_softpath_rectangle_opi:nnNnn {#1} {#2} }
\cs_new_protected:Npn \__draw_softpath_rectangle_opi:nnNnn #1#2#3#4#5
  { \driver_draw_rectangle:nnnn {#1} {#2} {#4} {#5} }
  \cs_new_protected:Npn \__draw_softpath_rectangle_opii:nn #1#2 { }
```

(*End definition for* `\__draw_softpath_close_op:nn` *and others.*)

⟨/initex | package⟩

6 l3draw-state implementation

⟨*initex | package⟩

⟨@@=draw⟩

`\g__draw_linewidth_dim` `\g__draw_inner_linewidth_dim`

Linewidth for strokes: global as the scope for this relies on the graphics state. The inner line width is used for places where two lines are used.

```
\dim_new:N \g__draw_linewidth_dim
\dim_new:N \g__draw_inner_linewidth_dim
```

(*End definition for* `\g__draw_linewidth_dim` *and* `\g__draw_inner_linewidth_dim`.)

`\l_draw_default_linewidth_dim`

A default: this is used at the start of every drawing.

```
\dim_new:N \l_draw_default_linewidth_dim
\dim_set:Nn \l_draw_default_linewidth_dim { 0.4pt }
```

(*End definition for* `\l_draw_default_linewidth_dim`. *This variable is documented on page* ??.)

`\draw_linewidth:n` `\draw_inner_linewidth:n`

Set the linewidth: we need a wrapper as this has to pass to the driver layer. The inner version is handled at the macro layer but is given a consistent interface here.

```
\cs_new_protected:Npn \draw_linewidth:n #1
  {
    \dim_gset:Nn \g__draw_linewidth_dim { \fp_to_dim:n {#1} }
    \driver_draw_linewidth:n \g__draw_linewidth_dim
  }
\cs_new_protected:Npn \draw_inner_linewidth:n #1
  { \dim_gset:Nn \g__draw_inner_linewidth_dim { \fp_to_dim:n {#1} }  }
```

(*End definition for* `\draw_linewidth:n` *and* `\draw_inner_linewidth:n`. *These functions are documented on page* 2.)

`\draw_miterlimit:n`

Pass through to the driver layer.

```
\cs_new_protected:Npn \draw_miterlimit:n #1
  { \driver_draw_miterlimit:n { \fp_to_dim:n {#1} } }
```

(*End definition for* `\draw_miterlimit:n`. *This function is documented on page* 2.)

`\draw_cap_butt:` `\draw_cap_rectangle:` `\draw_cap_round:` `\draw_evenodd_rule:` `\draw_nonzero_rule:` `\draw_join_bevel:` `\draw_join_miter:` `\draw_join_round:`

All straight wrappers.

```
\cs_new_protected:Npn \draw_cap_butt: { \driver_draw_cap_butt: }
\cs_new_protected:Npn \draw_cap_rectangle: { \driver_draw_cap_rectangle: }
\cs_new_protected:Npn \draw_cap_round: { \driver_draw_cap_round: }
\cs_new_protected:Npn \draw_evenodd_rule: { \driver_draw_evenodd_rule: }
\cs_new_protected:Npn \draw_nonzero_rule: { \driver_draw_nonzero_rule: }
\cs_new_protected:Npn \draw_join_bevel: { \driver_draw_join_bevel: }
\cs_new_protected:Npn \draw_join_miter: { \driver_draw_join_miter: }
\cs_new_protected:Npn \draw_join_round: { \driver_draw_join_round: }
```

(*End definition for* `\draw_cap_butt:` *and others. These functions are documented on page 2.*)

`\l__draw_color_tmp_tl` Scratch space.

```
\tl_new:N \l__draw_color_tmp_tl
```

(*End definition for* `\l__draw_color_tmp_tl`.)

`\g__draw_fill_color_tl` `\g__draw_stroke_color_tl` For tracking.

```
\tl_new:N \g__draw_fill_color_tl
\tl_new:N \g__draw_stroke_color_tl
```

(*End definition for* `\g__draw_fill_color_tl` *and* `\g__draw_stroke_color_tl`.)

`\draw_color:n` `\draw_color_fill:n` `\draw_color_stroke:n` `\__draw_color:nn` `\__draw_color_aux:nn` `\__draw_color_aux:Vn` `\__draw_color:nw` `\__draw_select_cmyk:nw` `\__draw_select_gray:nw` `\__draw_select_rgb:nw` `\__draw_split_select:nw` Much the same as for core color support but calling the relevant driver-level function.

```
\cs_new_protected:Npn \draw_color:n #1
  { \__draw_color:nn { } {#1} }
\cs_new_protected:Npn \draw_color_fill:n #1
  { \__draw_color:nn { fill } {#1} }
\cs_new_protected:Npn \draw_color_stroke:n #1
  { \__draw_color:nn { stroke } {#1} }
\cs_new_protected:Npn \__draw_color:nn #1#2
  {
    \color_parse:nN {#2} \l__draw_color_tmp_tl
    \tl_if_blank:nTF {#1}
      {
        \tl_gset_eq:NN \g__draw_fill_color_tl \l__draw_color_tmp_tl
        \tl_gset_eq:NN \g__draw_stroke_color_tl \l__draw_color_tmp_tl
        \__draw_color_aux:Vn \l__draw_color_tmp_tl { color }
      }
      {
        \tl_gset_eq:cN { g__draw_ #1 _color_tl } \l__draw_color_tmp_tl
        \__draw_color_aux:Vn \l__draw_color_tmp_tl { #1 }
      }
  }
\cs_new_protected:Npn \__draw_color_aux:nn #1#2
  { \__draw_color:nw {#2} #1 \q_stop }
\cs_generate_variant:Nn \__draw_color_aux:nn { V }
\cs_new_protected:Npn \__draw_color:nw #1#2 ~ #3 \q_stop
  { \use:c { __draw_color_ #2 :nw } {#1} #3 \q_stop }
\cs_new_protected:Npn \__draw_color_cmyk:nw #1#2 ~ #3 ~ #4 ~ #5 \q_stop
  { \use:c { driver_draw_ #1 _cmyk:nnnn } {#2} {#3} {#4} {#5} }
\cs_new_protected:Npn \__draw_color_gray:nw #1#2 \q_stop
  { \use:c { driver_draw_ #1 _gray:n } {#2} }
\cs_new_protected:Npn \__draw_color_rgb:nw #1#2 ~ #3 ~ #4 \q_stop
  { \use:c { driver_draw_ #1 _rgb:nnn } {#2} {#3} {#4} }
\cs_new_protected:Npn \__draw_color_spot:nw #1#2 ~ #3 \q_stop
  { \use:c { driver_draw_ #1 _spot:nn } {#2} {#3} }
```

(*End definition for* `\draw_color:n` *and others. These functions are documented on page 6.*)

```
⟨/initex | package⟩
```

7 l3draw-transforms implementation

```
1185 ⟨*initex | package⟩
1186 ⟨@@=draw⟩
```

`\l__draw_transformcm_active_bool` An internal flag to avoid redundant calculations.

```
1187 \bool_new:N \l__draw_transformcm_active_bool
```

(*End definition for* `\l__draw_transformcm_active_bool`.)

`\l__draw_transformcm_aa_fp` `\l__draw_transformcm_ab_fp` `\l__draw_transformcm_ba_fp` `\l__draw_transformcm_xshift_dim` `\l__draw_transformcm_yshift_dim` The active matrix itself.

```
1188 \fp_new:N \l__draw_transformcm_aa_fp
1189 \fp_new:N \l__draw_transformcm_ab_fp
1190 \fp_new:N \l__draw_transformcm_ba_fp
1191 \fp_new:N \l__draw_transformcm_bb_fp
1192 \dim_new:N \l__draw_transformcm_xshift_dim
1193 \dim_new:N \l__draw_transformcm_yshift_dim
```

(*End definition for* `\l__draw_transformcm_aa_fp` *and others.*)

`\draw_transform_reset:` Fast resetting.

```
1194 \cs_new_protected:Npn \draw_transform_reset:
1195   {
1196     \fp_set:Nn \l__draw_transformcm_aa_fp { 1}
1197     \fp_zero:N \l__draw_transformcm_ab_fp
1198     \fp_zero:N \l__draw_transformcm_ba_fp
1199     \fp_set:Nn \l__draw_transformcm_bb_fp { 1 }
1200     \dim_zero:N \l__draw_transformcm_xshift_dim
1201     \dim_zero:N \l__draw_transformcm_yshift_dim
1202   }
1203 \draw_transform_reset:
```

(*End definition for* `\draw_transform_reset:`. *This function is documented on page* 6.)

`\draw_transform:nnnnn` `\__draw_transform:nnnnnnn` Setting the transform matrix is straight-forward, with just a bit of expansion to sort out. With the mechanism active, the identity matrix is set.

```
1204 \cs_new_protected:Npn \draw_transform:nnnnn #1#2#3#4#5
1205   {
1206     \__draw_point_process:nn
1207       { \__draw_transform:nnnnnnn {#1} {#2} {#3} {#4} }
1208       {#5}
1209   }
1210 \cs_new_protected:Npn \__draw_transform:nnnnnnn #1#2#3#4#5#6
1211   {
1212     \fp_set:Nn \l__draw_transformcm_aa_fp {#1}
1213     \fp_set:Nn \l__draw_transformcm_ab_fp {#2}
1214     \fp_set:Nn \l__draw_transformcm_ba_fp {#3}
1215     \fp_set:Nn \l__draw_transformcm_bb_fp {#4}
1216     \dim_set:Nn \l__draw_transformcm_xshift_dim {#5}
1217     \dim_set:Nn \l__draw_transformcm_yshift_dim {#6}
1218     \bool_lazy_all:nTF
1219       {
1220         { \fp_compare_p:nNn \l__draw_transformcm_aa_fp = \c_one_fp }
1221         { \fp_compare_p:nNn \l__draw_transformcm_ab_fp = \c_zero_fp }
1222         { \fp_compare_p:nNn \l__draw_transformcm_ba_fp = \c_zero_fp }
```

```
        { \fp_compare_p:nNn \l__draw_transformcm_bb_fp = \c_one_fp }
      }
      { \bool_set_false:N \l__draw_transformcm_active_bool }
      { \bool_set_true:N \l__draw_transformcm_active_bool }
  }
```

(*End definition for* `\draw_transform:nnnnn` *and* `\__draw_transform:nnnnnnn`. *This function is documented on page* 7.)

`\draw_transform_concat:nnnnn`
`\__draw_transform_concat:nnnnnn`
`\__draw_transform_concat_aux:nnnnnn`

Much the same story for adding to an existing matrix. The part that is more complex is the calculations required: everything gets passed back to `\__draw_transform_set:nnnnnn`, with pre-expansion just in case there are *e.g* random values. The final step is `x`-type expanded as otherwise later values affect earlier ones.

```
\cs_new_protected:Npn \draw_transform_concat:nnnnn #1#2#3#4#5
  {
    \__draw_point_process:nn
      { \__draw_transform_concat:nnnnnn {#1} {#2} {#3} {#4} }
      {#5}
  }
\cs_new_protected:Npn \__draw_transform_concat:nnnnnn #1#2#3#4#5#6
  {
    \use:x
      {
        \__draw_transform_concat_aux:nnnnnn
          { \fp_eval:n {#1} }
          { \fp_eval:n {#2} }
          { \fp_eval:n {#3} }
          { \fp_eval:n {#4} }
          {#5}
          {#6}
      }
  }
\cs_new_protected:Npn \__draw_transform_concat_aux:nnnnnn #1#2#3#4#5#6
  {
    \use:x
      {
        \__draw_transform:nnnnnnn
          { #1 * \l__draw_transformcm_aa_fp + #2 * \l__draw_transformcm_ba_fp }
          { #1 * \l__draw_transformcm_ab_fp + #2 * \l__draw_transformcm_bb_fp }
          { #3 * \l__draw_transformcm_aa_fp + #4 * \l__draw_transformcm_ba_fp }
          { #3 * \l__draw_transformcm_ab_fp + #4 * \l__draw_transformcm_bb_fp }
          {
            \fp_to_dim:n
              {
                  \l__draw_transformcm_xshift_dim
                + \l__draw_transformcm_aa_fp * #5
                + \l__draw_transformcm_ba_fp * #6
              }
          }
          {
            \fp_to_dim:n
              {
                  \l__draw_transformcm_yshift_dim
                + \l__draw_transformcm_ab_fp * #5
```

```
                + \l__draw_transformcm_bb_fp * #6
              }
          }
        }
  }
```

(*End definition for* `\draw_transform_concat:nnnnn`, `\__draw_transform_concat:nnnnnn`, *and* `\__draw_transform_concat_aux:nnnnnn`. *This function is documented on page* 6.)

`\draw_transform_invert:`
`\__draw_transform_invert:n`
`\__draw_transform_invert:f`

Standard mathematics: calculate the inverse matrix and use that, then undo the shifts.

```
\cs_new_protected:Npn \draw_transform_invert:
  {
    \bool_if:NT \l__draw_transformcm_active_bool
      {
        \__draw_transform_invert:f
          {
            \fp_eval:n
              {
                1 /
                  (
                      \l__draw_transformcm_aa_fp * \l__draw_transformcm_bb_fp
                    - \l__draw_transformcm_ab_fp * \l__draw_transformcm_ba_fp
                  )
              }
          }
      }
    \dim_set:Nn \l__draw_transformcm_xshift_dim
      {
        \fp_to_dim:n
          {
            -\l__draw_transformcm_xshift_dim * \l__draw_transformcm_aa_fp
            -\l__draw_transformcm_yshift_dim * \l__draw_transformcm_ba_fp
          }
      }
    \dim_set:Nn \l__draw_transformcm_yshift_dim
      {
        \fp_to_dim:n
          {
            -\l__draw_transformcm_xshift_dim * \l__draw_transformcm_ab_fp
            -\l__draw_transformcm_yshift_dim * \l__draw_transformcm_bb_fp
          }
      }
  }
\cs_new_protected:Npn \__draw_transform_invert:n #1
  {
    \fp_set:Nn \l__draw_transformcm_aa_fp
      { \l__draw_transformcm_bb_fp * #1 }
    \fp_set:Nn \l__draw_transformcm_ab_fp
      { -\l__draw_transformcm_ab_fp * #1 }
    \fp_set:Nn \l__draw_transformcm_ba_fp
      { -\l__draw_transformcm_ba_fp * #1 }
    \fp_set:Nn \l__draw_transformcm_bb_fp
      { \l__draw_transformcm_aa_fp * #1 }
  }
\cs_generate_variant:Nn \__draw_transform_invert:n { f }
```

(*End definition for* `\draw_transform_invert:` *and* `\__draw_transform_invert:n`*. This function is documented on page 7.*)

`\draw_transform_triangle:nnn` Simple maths to move the canvas origin to `#1` and the two axes to `#2` and `#3`.

```
\cs_new_protected:Npn \draw_transform_triangle:nnn #1#2#3
  {
    \__draw_point_process:nnn
      {
        \__draw_point_process:nn
          { \__draw_tranform_triangle:nnnnnn }
          {#1}
      }
      {#2} {#3}
  }
\cs_new_protected:Npn \__draw_tranform_triangle:nnnnnn #1#2#3#4#5#6
  {
    \use:x
      {
        \__draw_transform:nnnnnnn
          { #3 - #1 }
          { #4 - #2 }
          { #5 - #1 }
          { #6 - #2 }
          {#1}
          {#2}
      }
  }
```

(*End definition for* `\draw_transform_triangle:nnn`*. This function is documented on page 7.*)

⟨/initex | package⟩

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

draw internal commands: