File I
Implementation

1 l3draw implementation

```
⟨*package⟩
⟨@@=draw⟩
\ProvidesExplPackage{l3draw}{2024-03-14}{}
  {L3 Experimental core drawing support}
```

1.1 Internal auxiliaries

\s__draw_mark
\s__draw_stop

Internal scan marks.

```
\scan_new:N \s__draw_mark
\scan_new:N \s__draw_stop
```

(*End of definition for* \s__draw_mark *and* \s__draw_stop.)

\q__draw_recursion_tail
\q__draw_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__draw_recursion_tail
\quark_new:N \q__draw_recursion_stop
```

(*End of definition for* \q__draw_recursion_tail *and* \q__draw_recursion_stop.)

__draw_if_recursion_tail_stop_do:Nn

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__draw_if_recursion_tail_stop_do:Nn
```

(*End of definition for* __draw_if_recursion_tail_stop_do:Nn.)

Everything else is in the sub-files!

```
⟨/package⟩
```

2 l3draw-boxes implementation

```
⟨*package⟩
⟨@@=draw⟩
```

Inserting boxes requires us to "interrupt" the drawing state, so is closely linked to scoping. At the same time, there are a few additional features required to make text work in a flexible way.

\l__draw_tmp_box

```
\box_new:N \l__draw_tmp_box
```

(*End of definition for* \l__draw_tmp_box.)

\draw_box_use:N
\draw_box_use:Nn
__draw_box_use:nNnnnnn
__draw_box_use:Nnnnn

Before inserting a box, we need to make sure that the bounding box is being updated correctly. As drawings track transformations as a whole, rather than as separate operations, we do the insertion using an almost-raw matrix. The process is split into two so that coffins are also supported.

```
\cs_new_protected:Npn \draw_box_use:N #1
  {
```

```
    \__draw_box_use:Nnnnnnn #1
      { 0pt } { -\box_dp:N #1 } { \box_wd:N #1 } { \box_ht:N #1 }
  }
\cs_new_protected:Npn \draw_box_use:Nn #1#2
  {
    \__draw_box_use:nNnnnn {#2} #1
      { 0pt } { -\box_dp:N #1 } { \box_wd:N #1 } { \box_ht:N #1 }
  }
\cs_new_protected:Npn \__draw_box_use:nNnnnn #1#2#3#4#5#6
  {
    \draw_scope_begin:
      \draw_transform_shift:n {#1}
      \__draw_box_use:Nnnnnnn #2 {#3} {#4} {#5} {#6}
    \draw_scope_end:
  }
\cs_new_protected:Npn \__draw_box_use:Nnnnnnn #1#2#3#4#5
  {
    \bool_if:NT \l_draw_bb_update_bool
      {
        \__draw_point_process:nn
          { \__draw_path_update_limits:nn }
          { \draw_point_transform:n { #2 , #3 } }
        \__draw_point_process:nn
          { \__draw_path_update_limits:nn }
          { \draw_point_transform:n { #4 , #3 } }
        \__draw_point_process:nn
          { \__draw_path_update_limits:nn }
          { \draw_point_transform:n { #4 , #5 } }
        \__draw_point_process:nn
          { \__draw_path_update_limits:nn }
          { \draw_point_transform:n { #2 , #5 } }
      }
    \group_begin:
      \hbox_set:Nn \l__draw_tmp_box
        {
          \use:e
            {
              \__draw_backend_box_use:Nnnnn #1
                { \fp_use:N \l__draw_matrix_a_fp }
                { \fp_use:N \l__draw_matrix_b_fp }
                { \fp_use:N \l__draw_matrix_c_fp }
                { \fp_use:N \l__draw_matrix_d_fp }
            }
        }
      \hbox_set:Nn \l__draw_tmp_box
        {
          \__kernel_kern:n { \l__draw_xshift_dim }
          \box_move_up:nn { \l__draw_yshift_dim }
            { \box_use_drop:N \l__draw_tmp_box }
        }
      \box_set_ht:Nn \l__draw_tmp_box { 0pt }
      \box_set_dp:Nn \l__draw_tmp_box { 0pt }
      \box_set_wd:Nn \l__draw_tmp_box { 0pt }
      \box_use_drop:N \l__draw_tmp_box
```

```
    \group_end:
  }
```

(*End of definition for* `\draw_box_use:N` *and others. These functions are documented on page* **??**.)

`\draw_coffin_use:Nnn`
`\draw_coffin_use:Nnnn`
`\__draw_coffin_use:nNnn`

Slightly more than a shortcut: we have to allow for the fact that coffins have no apparent width before the reference point.

```
\cs_new_protected:Npn \draw_coffin_use:Nnn #1#2#3
  {
    \__draw_coffin_use:nNnn { \__draw_box_use:Nnnnnnn }
      #1 {#2} {#3}
  }
\cs_new_protected:Npn \draw_coffin_use:Nnnn #1#2#3#4
  {
    \__draw_coffin_use:nNnn { \__draw_box_use:nNnnnn {#4} }
      #1 {#2} {#3}
  }
\cs_new_protected:Npn \__draw_coffin_use:nNnn #1#2#3#4
  {
    \group_begin:
      \hbox_set:Nn \l__draw_tmp_box
        { \coffin_typeset:Nnnnn #2 {#3} {#4} { 0pt } { 0pt } }
      #1 \l__draw_tmp_box
        { \box_wd:N \l__draw_tmp_box - \coffin_wd:N #2 }
        { -\box_dp:N \l__draw_tmp_box }
        { \box_wd:N \l__draw_tmp_box }
        { \box_ht:N \l__draw_tmp_box }
    \group_end:
  }
```

(*End of definition for* `\draw_coffin_use:Nnn`*,* `\draw_coffin_use:Nnnn`*, and* `\__draw_coffin_use:nNnn`*. These functions are documented on page* **??**.)

```
⟨/package⟩
```

3 l3draw-layers implementation

```
⟨*package⟩
⟨@@=draw⟩
```

3.1 User interface

`\draw_layer_new:n`

```
\cs_new_protected:Npn \draw_layer_new:n #1
  {
    \str_if_eq:nnTF {#1} { main }
      { \msg_error:nnn { draw } { main-reserved } }
      {
        \box_new:c { g__draw_layer_ #1 _box }
        \box_new:c { l__draw_layer_ #1 _box }
      }
  }
```

(*End of definition for* `\draw_layer_new:n`*. This function is documented on page* **??**.)

\l__draw_layer_tl

The name of the current layer: we start off with `main`.

```
\tl_new:N \l__draw_layer_tl
\tl_set:Nn \l__draw_layer_tl { main }
```

(*End of definition for* `\l__draw_layer_tl`*.*)

\l__draw_layer_close_bool

Used to track if a layer needs to be closed.

```
\bool_new:N \l__draw_layer_close_bool
```

(*End of definition for* `\l__draw_layer_close_bool`*.*)

\l_draw_layers_clist
\g__draw_layers_clist

The list of layers to use starts off with just the `main` one.

```
\clist_new:N \l_draw_layers_clist
\clist_set:Nn \l_draw_layers_clist { main }
\clist_new:N \g__draw_layers_clist
```

(*End of definition for* `\l_draw_layers_clist` *and* `\g__draw_layers_clist`*. This variable is documented on page* **??***.*)

\draw_layer_begin:n
\draw_layer_end:

Layers may be called multiple times and have to work when nested. That drives a bit of grouping to get everything in order. Layers have to be zero width, so they get set as we go along.

```
\cs_new_protected:Npn \draw_layer_begin:n #1
  {
    \group_begin:
      \box_if_exist:cTF { g__draw_layer_ #1 _box }
        {
          \str_if_eq:VnTF \l__draw_layer_tl {#1}
            { \bool_set_false:N \l__draw_layer_close_bool }
            {
              \bool_set_true:N \l__draw_layer_close_bool
              \tl_set:Nn \l__draw_layer_tl {#1}
              \box_gset_wd:cn { g__draw_layer_ #1 _box } { 0pt }
              \hbox_gset:cw { g__draw_layer_ #1 _box }
                \box_use_drop:c { g__draw_layer_ #1 _box }
                \group_begin:
            }
          \draw_linewidth:n { \l_draw_default_linewidth_dim }
        }
        {
          \str_if_eq:nnTF {#1} { main }
            { \msg_error:nnn { draw } { unknown-layer } {#1} }
            { \msg_error:nnn { draw } { main-layer } }
        }
  }
\cs_new_protected:Npn \draw_layer_end:
  {
      \bool_if:NT \l__draw_layer_close_bool
        {
            \group_end:
          \hbox_gset_end:
        }
    \group_end:
  }
```

(*End of definition for* `\draw_layer_begin:n` *and* `\draw_layer_end:`*. These functions are documented on page* **??***.*)

3.2 Internal cross-links

__draw_layers_insert: The main layer is special, otherwise just dump the layer box inside a scope.

```
\cs_new_protected:Npn \__draw_layers_insert:
  {
    \clist_map_inline:Nn \l_draw_layers_clist
      {
        \str_if_eq:nnTF {##1} { main }
          {
            \box_set_wd:Nn \l__draw_layer_main_box { 0pt }
            \box_use_drop:N \l__draw_layer_main_box
          }
          {
            \__draw_backend_scope_begin:
            \box_gset_wd:cn { g__draw_layer_ ##1 _box } { 0pt }
            \box_use_drop:c { g__draw_layer_ ##1 _box }
            \__draw_backend_scope_end:
          }
      }
  }
```

(*End of definition for* __draw_layers_insert:.)

__draw_layers_save:
__draw_layers_restore:

Simple save/restore functions.

```
\cs_new_protected:Npn \__draw_layers_save:
  {
    \clist_map_inline:Nn \l_draw_layers_clist
      {
        \str_if_eq:nnF {##1} { main }
          {
            \box_set_eq:cc { l__draw_layer_ ##1 _box }
              { g__draw_layer_ ##1 _box }
          }
      }
  }
\cs_new_protected:Npn \__draw_layers_restore:
  {
    \clist_map_inline:Nn \l_draw_layers_clist
      {
        \str_if_eq:nnF {##1} { main }
          {
            \box_gset_eq:cc { g__draw_layer_ ##1 _box }
              { l__draw_layer_ ##1 _box }
          }
      }
  }
```

(*End of definition for* __draw_layers_save: *and* __draw_layers_restore:.)

```
\msg_new:nnnn { draw } { main-layer }
  { Material~cannot~be~added~to~'main'~layer. }
  { The~main~layer~may~only~be~accessed~at~the~top~level. }
\msg_new:nnn { draw } { main-reserved }
  { The~'main'~layer~is~reserved. }
\msg_new:nnnn { draw } { unknown-layer }
```

```
  { Layer~'#1'~has~not~been~created. }
  { You~have~tried~to~use~layer~'#1',~but~it~was~never~set~up. }
% \end{macrocode}
%
%    \begin{macrocode}
⟨/package⟩
```

4 l3draw-paths implementation

```
⟨*package⟩
⟨@@=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 of definition for* `\l__draw_path_tmp_tl`*,* `\l__draw_path_tmpa_fp`*, and* `\l__draw_path_tmpb_fp`*.*)

4.1 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 of 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 of 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_bb_update_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 of 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 of definition for* `\__draw_path_update_last:nn`.)

4.2 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 of 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 of definition for* \l__draw_corner_arc_bool*.*)

\draw_path_corner_arc:nn

Calculate the arcs, check they are non-zero.

```
\cs_new_protected:Npn \draw_path_corner_arc:nn #1#2
  {
    \dim_set:Nn \l__draw_corner_xarc_dim { \fp_to_dim:n {#1} }
    \dim_set:Nn \l__draw_corner_yarc_dim { \fp_to_dim:n {#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 of definition for* \draw_path_corner_arc:nn*. This function is documented on page* **??***.*)

__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 of definition for* __draw_path_mark_corner:*.*)

4.3 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:nnnn
      {
        \__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 of definition for* `\draw_path_moveto:n` *and others. These functions are documented on page* **??**.)

`\draw_path_close:`

A simple wrapper.

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

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

4.4 Canvas path constructions

`\draw_path_canvas_moveto:n`
`\draw_path_canvas_lineto:n`
`\draw_path_canvas_curveto:nnn`

Operations with no application of the transformation matrix.

```
\cs_new_protected:Npn \draw_path_canvas_moveto:n #1
  { \__draw_point_process:nn { \__draw_path_moveto:nn } {#1} }
\cs_new_protected:Npn \draw_path_canvas_lineto:n #1
  { \__draw_point_process:nn { \__draw_path_lineto:nn } {#1} }
\cs_new_protected:Npn \draw_path_canvas_curveto:nnn #1#2#3
  {
    \__draw_point_process:nnnn
      {
        \__draw_path_mark_corner:
        \__draw_path_curveto:nnnnnn
      }
```

```
      {#1} {#2} {#3}
  }
```

(*End of definition for* `\draw_path_canvas_moveto:n`*,* `\draw_path_canvas_lineto:n`*, and* `\draw_path_canvas_curveto:nnn`*. These functions are documented on page* **??***.*)

4.5 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 $(x_\mathrm{c}, y_\mathrm{c})$. The two required control points are then

$$x_1 = \frac{1}{3}x_\mathrm{s} + \frac{2}{3}x_\mathrm{c} \quad y_1 = \frac{1}{3}y_\mathrm{s} + \frac{2}{3}y_\mathrm{c}$$

and

$$x_2 = \frac{1}{3}x_\mathrm{e} + \frac{2}{3}x_\mathrm{c} \quad x_2 = \frac{1}{3}y_\mathrm{e} + \frac{2}{3}y_\mathrm{c}$$

using the start (last) point $(x_\mathrm{s}, y_\mathrm{s})$ and the end point $(x_\mathrm{s}, y_\mathrm{s})$.

```
\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:e
      {
        \__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 of definition for* `\draw_path_curveto:nn` *and others. This function is documented on page* **??**.)

`\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:enenNnn`
`\__draw_path_arc_auxi:eennNnn`
`\__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 Bézier 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:e
      {
        \__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:eennNnn
              { \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:eennNnn
              { \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:enenNnn
      { \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:e
      {
        \__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 { ene , ee }
```

We can now calculate the required points. As everything here is non-expandable, that is best done by using e-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 {#7} {#8} { #1 #4 90 } }
      }
    \__draw_point_process:nnn
      { \__draw_path_arc_auxiv:nnnn }
      {
        \draw_point_transform:n
          { \draw_point_polar:nnn {#5} {#6} {#1} }
      }
      {
        \draw_point_transform:n
          { \draw_point_polar:nnn {#5} {#6} {#2} }
      }
    \__draw_point_process:nn
      { \__draw_path_arc_auxv:nn }
      {
        \__draw_point_transform_noshift:n
          { \draw_point_polar:nnn {#7} {#8} { #2 #4 -90 } }
      }
    \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:Ne \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 of definition for* `\draw_path_arc:nnn` *and others. These functions are documented on page* **??**.)

`\draw_path_arc_axes:nnnn`

A simple wrapper.

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

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

`\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:nnnn
      { \__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:e
      {
        \__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 of 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 of definition for* `\draw_path_circle:nn`. *This function is documented on page* **??**.)

4.6 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
  {
    \bool_lazy_or:nnTF
      { \l__draw_corner_arc_bool }
      { \l__draw_matrix_active_bool }
      {
        \__draw_point_process:nnn \__draw_path_rectangle_rounded:nnnn
```

```
          {#1} {#2}
      }
      {
        \__draw_point_process:nnn \__draw_path_rectangle:nnnn
          { (#1) + ( \l__draw_xshift_dim , \l__draw_yshift_dim ) }
          { #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 of definition for* `\draw_path_rectangle:nn`*,* `\__draw_path_rectangle:nnnn`*, and* `\__draw_path_rectangle_rounded:nnnn`*. This function is documented on page* **??**.)

\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 of definition for* `\draw_path_rectangle_corners:nn` *and* `\__draw_path_rectangle_corners:nnnn`*. This function is documented on page* **??**.)

4.7 Grids

\draw_path_grid:nnnn
__draw_path_grid_auxi:nnnnnn
__draw_path_grid_auxi:eennnn
__draw_path_grid_auxii:nnnnnn
__draw_path_grid_auxiii:nnnnnn
__draw_path_grid_auxiiii:eennnn
__draw_path_grid_auxiv:nnnnnnnn
__draw_path_grid_auxiv:eennnnnn

The main complexity here is lining up the grid correctly. To keep it simple, we tidy up the argument ordering first.

```
\cs_new_protected:Npn \draw_path_grid:nnnn #1#2#3#4
  {
    \__draw_point_process:nnn
      {
        \__draw_path_grid_auxi:eennnn
          { \dim_abs:n {#1} }
          { \dim_abs:n {#2} }
      }
      {#3} {#4}
```

```
  }
\cs_new_protected:Npn \__draw_path_grid_auxi:nnnnnn #1#2#3#4#5#6
  {
    \dim_compare:nNnTF {#3} > {#5}
      { \__draw_path_grid_auxii:nnnnnn {#1} {#2} {#5} {#4} {#3} {#6} }
      { \__draw_path_grid_auxii:nnnnnn {#1} {#2} {#3} {#4} {#5} {#6} }
  }
\cs_generate_variant:Nn \__draw_path_grid_auxi:nnnnnn { ee }
\cs_new_protected:Npn \__draw_path_grid_auxii:nnnnnn #1#2#3#4#5#6
  {
    \dim_compare:nNnTF {#4} > {#6}
      { \__draw_path_grid_auxiii:nnnnnn {#1} {#2} {#3} {#6} {#5} {#4} }
      { \__draw_path_grid_auxiii:nnnnnn {#1} {#2} {#3} {#4} {#5} {#6} }
  }
\cs_new_protected:Npn \__draw_path_grid_auxiii:nnnnnn #1#2#3#4#5#6
  {
    \__draw_path_grid_auxiv:eennnnnn
      { \fp_to_dim:n { #1 * ceil(#3/(#1)) } }
      { \fp_to_dim:n { #2 * ceil(#4/(#2)) } }
      {#1} {#2} {#3} {#4} {#5} {#6}
  }
\cs_new_protected:Npn \__draw_path_grid_auxiv:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \dim_step_inline:nnnn
      {#1}
      {#3}
      {#7}
      {
        \draw_path_moveto:n { ##1 , #6 }
        \draw_path_lineto:n { ##1 , #8 }
      }
    \dim_step_inline:nnnn
      {#2}
      {#4}
      {#8}
      {
        \draw_path_moveto:n { #5 , ##1 }
        \draw_path_lineto:n { #7 , ##1 }
      }
  }
\cs_generate_variant:Nn \__draw_path_grid_auxiv:nnnnnnnn { ee }
```

(*End of definition for* `\draw_path_grid:nnnn` *and others. This function is documented on page* **??**.)

4.8 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 of 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_clear_bool

Actions handled at the macro layer.

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

(*End of definition for* `\l__draw_path_use_clear_bool`.)

\draw_path_use:n
\draw_path_use_clear:n
\draw_path_replace_bb:
__draw_path_replace_bb:NnN
__draw_path_use:n
__draw_path_use_action_draw:
__draw_path_use_action_fillstroke:
__draw_path_use_stroke_bb:
__draw_path_use_bb: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 } }
  }
\cs_new_protected:Npn \draw_path_replace_bb:
  {
    \__draw_path_replace_bb:NnN x { max } +
    \__draw_path_replace_bb:NnN y { max } +
    \__draw_path_replace_bb:NnN x { min } -
    \__draw_path_replace_bb:NnN y { min } -
    \__draw_softpath_clear:
    \__draw_path_reset_limits:
  }
\cs_new_protected:Npn \__draw_path_replace_bb:NnN #1#2#3
  {
    \dim_gset:cn { g__draw_ #1#2 _dim }
      {
          \dim_use:c { g__draw_path_ #1#2 _dim }
        #3 0.5 \g__draw_linewidth_dim
      }
  }
```

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 : }
            { \msg_error:nnn { draw } { invalid-path-action } {##1} }
        }
      }
    \__draw_softpath_round_corners:
    \bool_lazy_and:nnT
      { \l_draw_bb_update_bool }
      { \l__draw_path_use_stroke_bool }
      { \__draw_path_use_stroke_bb: }
    \__draw_softpath_use:
    \bool_if:NT \l__draw_path_use_clip_bool
      {
        \__draw_backend_clip:
        \bool_set_false:N \l_draw_bb_update_bool
        \bool_lazy_or:nnF
          { \l__draw_path_use_fill_bool }
          { \l__draw_path_use_stroke_bool }
          { \__draw_backend_discardpath: }
      }
    \bool_lazy_or:nnT
      { \l__draw_path_use_fill_bool }
      { \l__draw_path_use_stroke_bool }
      {
        \use:c
          {
            __draw_backend_
            \bool_if:NT \l__draw_path_use_fill_bool { fill }
            \bool_if:NT \l__draw_path_use_stroke_bool { stroke }
            :
          }
      }
    \bool_if:NT \l__draw_path_use_clear_bool
      {
        \__draw_softpath_clear:
        \__draw_path_reset_limits:
      }
  }
\cs_new_protected:Npn \__draw_path_use_action_draw:
  {
    \bool_set_true:N \l__draw_path_use_stroke_bool
  }
\cs_new_protected:Npn \__draw_path_use_action_fillstroke:
  {
    \bool_set_true:N \l__draw_path_use_fill_bool
    \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_bb:NnN x { max } +
    \__draw_path_use_bb:NnN y { max } +
```

```
    \__draw_path_use_bb:NnN x { min } -
    \__draw_path_use_bb:NnN y { min } -
  }
\cs_new_protected:Npn \__draw_path_use_bb: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 of definition for* `\draw_path_use:n` *and others. These functions are documented on page* **??**.)

4.9 Scoping paths

\l__draw_path_lastx_dim
\l__draw_path_lasty_dim
\l__draw_path_xmax_dim
\l__draw_path_xmin_dim
\l__draw_path_ymax_dim
\l__draw_path_ymin_dim
\l__draw_softpath_corners_bool

Local storage for global data. There is already a `\l__draw_softpath_main_tl` for path manipulation, so we can reuse that (it is always grouped when the path is being reconstructed).

```
\dim_new:N \l__draw_path_lastx_dim
\dim_new:N \l__draw_path_lasty_dim
\dim_new:N \l__draw_path_xmax_dim
\dim_new:N \l__draw_path_xmin_dim
\dim_new:N \l__draw_path_ymax_dim
\dim_new:N \l__draw_path_ymin_dim
\dim_new:N \l__draw_softpath_lastx_dim
\dim_new:N \l__draw_softpath_lasty_dim
\bool_new:N \l__draw_softpath_corners_bool
```

(*End of definition for* `\l__draw_path_lastx_dim` *and others.*)

`\draw_path_scope_begin:`
`\draw_path_scope_end:`

Scoping a path is a bit more involved, largely as there are a number of variables to keep hold of.

```
\cs_new_protected:Npn \draw_path_scope_begin:
  {
    \group_begin:
      \dim_set_eq:NN \l__draw_path_lastx_dim \g__draw_path_lastx_dim
      \dim_set_eq:NN \l__draw_path_lasty_dim \g__draw_path_lasty_dim
      \dim_set_eq:NN \l__draw_path_xmax_dim \g__draw_path_xmax_dim
      \dim_set_eq:NN \l__draw_path_xmin_dim \g__draw_path_xmin_dim
      \dim_set_eq:NN \l__draw_path_ymax_dim \g__draw_path_ymax_dim
      \dim_set_eq:NN \l__draw_path_ymin_dim \g__draw_path_ymin_dim
      \dim_set_eq:NN \l__draw_softpath_lastx_dim \g__draw_softpath_lastx_dim
      \dim_set_eq:NN \l__draw_softpath_lasty_dim \g__draw_softpath_lasty_dim
      \__draw_path_reset_limits:
      \__draw_softpath_save:
  }
```

```
\cs_new_protected:Npn \draw_path_scope_end:
  {
      \__draw_softpath_restore:
      \dim_gset_eq:NN \g__draw_softpath_lastx_dim \l__draw_softpath_lastx_dim
      \dim_gset_eq:NN \g__draw_softpath_lasty_dim \l__draw_softpath_lasty_dim
      \dim_gset_eq:NN \g__draw_path_xmax_dim \l__draw_path_xmax_dim
      \dim_gset_eq:NN \g__draw_path_xmin_dim \l__draw_path_xmin_dim
      \dim_gset_eq:NN \g__draw_path_ymax_dim \l__draw_path_ymax_dim
      \dim_gset_eq:NN \g__draw_path_ymin_dim \l__draw_path_ymin_dim
      \dim_gset_eq:NN \g__draw_path_lastx_dim \l__draw_path_lastx_dim
      \dim_gset_eq:NN \g__draw_path_lasty_dim \l__draw_path_lasty_dim
    \group_end:
  }
```

(*End of definition for* `\draw_path_scope_begin:` *and* `\draw_path_scope_end:`*. These functions are documented on page* **??**.)

4.10 Messages

```
\msg_new:nnnn { draw } { invalid-path-action }
  { Invalid~action~'#1'~for~path. }
  { Paths~can~be~used~with~actions~'draw',~'clip',~'fill'~or~'stroke'. }
%     \end{macrocode}
%
%    \begin{macrocode}
⟨/package⟩
```

5 l3draw-points implementation

```
⟨*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`.
- `\pgfpointadd`, `\pgfpointdiff`, `\pgfpointscale`: Can be given explicitly.
- `\pgfextractx`, `\pgfextracty`: Available by applying `\use_i:nn`/`\use_ii:nn` or similar to the `e`-type expansion of a point expression.
- `\pgfgetlastxy`: Unused in the entire `pgf` core, may be emulated by `e`-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.

5.1 Support functions

__draw_point_process:nn
__draw_point_process_auxi:nn
__draw_point_process_auxi:en
__draw_point_process_auxii:nw
__draw_point_process:nnn
__draw_point_process_auxiii:nnn
__draw_point_process_auxiii:een
__draw_point_process_auxiv:nw
__draw_point_process:nnnn
__draw_point_process_auxv:nnnn
__draw_point_process_auxv:eeen
__draw_point_process_auxvi:nw
__draw_point_process:nnnnn
__draw_point_process_auxvii:nnnnn
__draw_point_process_auxvii:eeeen
__draw_point_process_auxviii: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:en
      { \draw_point:n {#2} }
      {#1}
  }
\cs_new:Npn \__draw_point_process_auxi:nn #1#2
  { \__draw_point_process_auxii:nw {#2} #1 \s__draw_stop }
\cs_generate_variant:Nn \__draw_point_process_auxi:nn { e }
\cs_new:Npn \__draw_point_process_auxii:nw #1 #2 , #3 \s__draw_stop
  { #1 {#2} {#3} }
\cs_new:Npn \__draw_point_process:nnn #1#2#3
  {
    \__draw_point_process_auxiii:een
      { \draw_point:n {#2} }
      { \draw_point:n {#3} }
      {#1}
  }
\cs_new:Npn \__draw_point_process_auxiii:nnn #1#2#3
  { \__draw_point_process_auxiv:nw {#3} #1 \s__draw_mark #2 \s__draw_stop }
\cs_generate_variant:Nn \__draw_point_process_auxiii:nnn { ee }
\cs_new:Npn \__draw_point_process_auxiv:nw #1 #2 , #3 \s__draw_mark #4 , #5 \s__draw_stop
  { #1 {#2} {#3} {#4} {#5} }
\cs_new:Npn \__draw_point_process:nnnn #1#2#3#4
  {
    \__draw_point_process_auxv:eeen
      { \draw_point:n {#2} }
      { \draw_point:n {#3} }
      { \draw_point:n {#4} }
      {#1}
  }
\cs_new:Npn \__draw_point_process_auxv:nnnn #1#2#3#4
  { \__draw_point_process_auxvi:nw {#4} #1 \s__draw_mark #2 \s__draw_mark #3 \s__draw_stop }
\cs_generate_variant:Nn \__draw_point_process_auxv:nnnn { eee }
\cs_new:Npn \__draw_point_process_auxvi:nw
  #1 #2 , #3 \s__draw_mark #4 , #5 \s__draw_mark #6 , #7 \s__draw_stop
  { #1 {#2} {#3} {#4} {#5} {#6} {#7} }
\cs_new:Npn \__draw_point_process:nnnnn #1#2#3#4#5
  {
    \__draw_point_process_auxvii:eeeen
      { \draw_point:n {#2} }
      { \draw_point:n {#3} }
```

```
    { \draw_point:n {#4} }
    { \draw_point:n {#5} }
    {#1}
  }
\cs_new:Npn \__draw_point_process_auxvii:nnnnn #1#2#3#4#5
  {
    \__draw_point_process_auxviii:nw
      {#5} #1 \s__draw_mark #2 \s__draw_mark #3 \s__draw_mark #4 \s__draw_stop
  }
\cs_generate_variant:Nn \__draw_point_process_auxvii:nnnnn { eeee }
\cs_new:Npn \__draw_point_process_auxviii:nw
  #1 #2 , #3 \s__draw_mark #4 , #5 \s__draw_mark #6 , #7 \s__draw_mark #8 , #9 \s__draw_stop
  { #1 {#2} {#3} {#4} {#5} {#6} {#7} {#8} {#9} }
```

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

5.2 Basic points

`\draw_point:n`
`\__draw_point_to_dim:n`
`\__draw_point_to_dim:e`
`\__draw_point_to_dim:w`

Co-ordinates are always returned as two dimensions.

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

5.3 Polar co-ordinates

`\draw_point_polar:nn`
`\draw_point_polar:nnn`
`\__draw_draw_polar:nnn`
`\__draw_draw_polar:enn`

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.

```
\cs_new:Npn \draw_point_polar:nn #1#2
  { \draw_point_polar:nnn {#1} {#1} {#2} }
\cs_new:Npn \draw_point_polar:nnn #1#2#3
  { \__draw_draw_polar:enn { \fp_eval:n {#3} } {#1} {#2} }
\cs_new:Npn \__draw_draw_polar:nnn #1#2#3
  { \draw_point:n { cosd(#1) * (#2) , sind(#1) * (#3) } }
\cs_generate_variant:Nn \__draw_draw_polar:nnn { e }
```

5.4 Point expression arithmetic

These functions all take point expressions as arguments.

`\draw_point_unit_vector:n`
`\__draw_point_unit_vector:nn`
`\__draw_point_unit_vector:nnn`
`\__draw_point_unit_vector:enn`

The outcome is the normalised vector from $(0,0)$ in the direction of the point, *i.e.*

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

except where the length is zero, in which case a vertical vector is returned.

```
\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_unit_vector:nnn
      { \fp_eval:n { (sqrt(#1 * #1 + #2 * #2)) } }
      {#1} {#2}
  }
\cs_new:Npn \__draw_point_unit_vector:nnn #1#2#3
  {
    \fp_compare:nNnTF {#1} = \c_zero_fp
      { 0pt, 1pt }
      {
        \draw_point:n
          { ( #2 , #3 ) / #1 }
      }
  }
\cs_generate_variant:Nn \__draw_point_unit_vector:nnn { e }
```

5.5 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:eeeeee

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

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

and

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

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:nnnnn
      { \__draw_point_intersect_lines:nnnnnnnn }
      {#1} {#2} {#3} {#4}
  }
```

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

#1 x_1

#2 y_1

#3 x_2

#4 y_2

#5 x_3

#6 y_3

#7 x_4

#8 y_4

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:eeeeee
      { \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:n
      {
        ( #2 * #3 - #1 * #4 , #2 * #5 - #1 * #6 )
          / ( #4 * #5 - #6 * #3 )
      }
  }
\cs_generate_variant:Nn \__draw_point_intersect_lines_aux:nnnnnn { eeeeee }
```

\draw_point_intersect_circles:nnnnn
__draw_point_intersect_circles_auxi:nnnnnnn
_draw_point_intersect_circles_auxii:nnnnnnn
_draw_point_intersect_circles_auxii:eennnnn
draw_point_intersect_circles_auxiii:nnnnnnn
draw_point_intersect_circles_auxiii:eennnnn
draw_point_intersect_circles_auxiv:nnnnnnnn
draw_point_intersect_circles_auxiv:ennnnnnn
draw_point_intersect_circles_auxv:nnnnnnnnn
draw_point_intersect_circles_auxv:eennnnnnn
draw_point_intersect_circles_auxvi:nnnnnnnn
draw_point_intersect_circles_auxvi:ennnnnnn
draw_point_intersect_circles_auxvii:nnnnnnn
draw_point_intersect_circles_auxvii:eeennnn

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

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

in either

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

or

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

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:eennnnn
      { \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:eennnnn
      { \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 { ee }
\cs_new:Npn \__draw_point_intersect_circles_auxiii:nnnnnnn #1#2#3#4#5#6#7
  {
    \__draw_point_intersect_circles_auxiv:ennnnnnn
      { \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 { ee }
```

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:eennnnnnn
      { \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 { e }
\cs_new:Npn \__draw_point_intersect_circles_auxv:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_intersect_circles_auxvi:ennnnnnn
      { \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 { ee }
```

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:eeennnn
      { \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 { e }
\cs_new:Npn \__draw_point_intersect_circles_auxvii:nnnnnnn #1#2#3#4#5#6#7
  {
    \draw_point:n
      { #6 + #4 * #1 + #2 * #3 * #5 , #7 + #5 * #1 + -1 * #2 * #3 * #4 }
  }
\cs_generate_variant:Nn \__draw_point_intersect_circles_auxvii:nnnnnnn { eee }
```

\draw_point_intersect_line_circle:nnnnn
w_point_intersect_line_circle_auxi:nnnnnnnn
_point_intersect_line_circle_auxii:nnnnnnnn
_point_intersect_line_circle_auxii:ennnnnn
point_intersect_line_circle_auxiii:nnnnnnnn
point_intersect_line_circle_auxiii:eeennnnn
_point_intersect_line_circle_auxiv:nnnnnnnn
_point_intersect_line_circle_auxiv:eennnnnn
draw_point_intersect_line_circle_auxv:nnnnn
draw_point_intersect_line_circle_auxv:ennnn

The intersection points P_1 and P_2 between a line joining points (x_1, y_1) and (x_2, y_2) and a circle with center (x_3, y_3) and radius r. We use the intermediate values

$$
\begin{aligned}
a &= (x_2 - x_1)^2 + (y_2 - y_1)^2 \\
b &= 2 \times ((x_2 - x_1) \times (x_1 - x_3) + (y_2 - y_1) \times (y_1 - y_3)) \\
c &= x_3^2 + y_3^2 + x_1^2 + y_1^2 - 2 \times (x_3 \times x_1 + y_3 \times y_1) - r^2 \\
d &= b^2 - 4 \times a \times c \\
\mu_1 &= \frac{-b + \sqrt{d}}{2 \times a} \\
\mu_2 &= \frac{-b - \sqrt{d}}{2 \times a}
\end{aligned}
$$

in either

$$
\begin{aligned}
P_{1x} &= x_1 + \mu_1 \times (x_2 - x_1) \\
P_{1y} &= y_1 + \mu_1 \times (y_2 - y_1)
\end{aligned}
$$

or

$$P_{2x} = x_1 + \mu_2 \times (x_2 - x_1)$$
$$P_{2y} = y_1 + \mu_2 \times (y_2 - y_1)$$

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_line_circle:nnnnn #1#2#3#4#5
  {
    \__draw_point_process:nnnn
      { \__draw_point_intersect_line_circle_auxi:nnnnnnnn {#4} {#5} }
      {#1} {#2} {#3}
  }
\cs_new:Npn \__draw_point_intersect_line_circle_auxi:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_line_circle_auxii:ennnnnnn
      { \fp_eval:n {#1} } {#3} {#4} {#5} {#6} {#7} {#8} {#2}
  }
```

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

`#1` r

`#2` x_1

`#3` y_1

`#4` x_2

`#5` y_2

`#6` x_3

`#7` y_3

`#8` n

Once we evaluate a, b and c, the co-ordinate (x_3, y_3) and r are no longer required: handy as we will need various intermediate values in the following.

```
\cs_new:Npn \__draw_point_intersect_line_circle_auxii:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_line_circle_auxiii:eeennnnn
      { \fp_eval:n { (#4-#2)*(#4-#2)+(#5-#3)*(#5-#3) } }
      { \fp_eval:n { 2*((#4-#2)*(#2-#6)+(#5-#3)*(#3-#7)) } }
      { \fp_eval:n { (#6*#6+#7*#7)+(#2*#2+#3*#3)-(2*(#6*#2+#7*#3))-(#1*#1) } }
      {#2} {#3} {#4} {#5} {#8}
  }
\cs_generate_variant:Nn \__draw_point_intersect_line_circle_auxii:nnnnnnnn { e }
```

then we can get $d = b^2 - 4 \times a \times c$ and the usage of n.

```
\cs_new:Npn \__draw_point_intersect_line_circle_auxiii:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_line_circle_auxiv:eennnnnn
      { \fp_eval:n {  #2 * #2 - 4 * #1 * #3 } }
      { \int_if_odd:nTF {#8} { 1 } { -1 } }
      {#1} {#2} {#4} {#5} {#6} {#7}
  }
\cs_generate_variant:Nn \__draw_point_intersect_line_circle_auxiii:nnnnnnnn { eee }
```

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 a

#2 b

#3 c

#4 d

#5 $\pm$(the usage of n)

#6 x_1

#7 y_1

#8 x_2

#9 y_2

There are some final pre-calculations, $\mu = \frac{-b \pm \sqrt{d}}{2 \times a}$ then, we can yield a result.

```
\cs_new:Npn \__draw_point_intersect_line_circle_auxiv:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \__draw_point_intersect_line_circle_auxv:ennnn
      { \fp_eval:n { (-1 * #4 + #2 * sqrt(#1)) / (2 * #3) } }
      {#5} {#6} {#7} {#8}
  }
\cs_generate_variant:Nn \__draw_point_intersect_line_circle_auxiv:nnnnnnnn { ee }
\cs_new:Npn \__draw_point_intersect_line_circle_auxv:nnnnn #1#2#3#4#5
  {
    \draw_point:n
      { #2 + #1 * (#4 - #2), #3 + #1 * (#5 - #3) }
  }
\cs_generate_variant:Nn \__draw_point_intersect_line_circle_auxv:nnnnn { e }
```

5.6 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:ennnn
__draw_point_interpolate_line_aux:nnnnnn
__draw_point_interpolate_line_aux:ennnnn

```
\cs_new:Npn \draw_point_interpolate_line:nnn #1#2#3
  {
    \__draw_point_process:nnn
      { \__draw_point_interpolate_line_aux:ennnn { \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:ennnnn { \fp_eval:n { 1 - #1 } }
      {#1} {#2} {#3} {#4} {#5}
  }
\cs_generate_variant:Nn \__draw_point_interpolate_line_aux:nnnnn { e }
\cs_new:Npn \__draw_point_interpolate_line_aux:nnnnnn #1#2#3#4#5#6
  { \draw_point:n { #2 * #3 + #1 * #5 , #2 * #4 + #1 * #6 } }
\cs_generate_variant:Nn \__draw_point_interpolate_line_aux:nnnnnn { e }
```

\draw_point_interpolate_distance:nnn
__draw_point_interpolate_distance:nnnnn
__draw_point_interpolate_distance:nnnnnn
__draw_point_interpolate_distance:ennnnn

Same idea but using the normalised length to obtain the scale factor. The start point is needed twice, so we force evaluation, but the end point is needed only the once.

```
\cs_new:Npn \draw_point_interpolate_distance:nnn #1#2#3
  {
    \__draw_point_process:nn
      { \__draw_point_interpolate_distance:nnnn {#1} {#3} }
      {#2}
  }
\cs_new:Npn \__draw_point_interpolate_distance:nnnn #1#2#3#4
  {
    \__draw_point_process:nn
      {
        \__draw_point_interpolate_distance:ennnn
          { \fp_eval:n {#1} } {#3} {#4}
      }
      { \draw_point_unit_vector:n { ( #2 ) - ( #3 , #4 ) } }
  }
\cs_new:Npn \__draw_point_interpolate_distance:nnnnn #1#2#3#4#5
  { \draw_point:n { #2 + #1 * #4 , #3 + #1 * #5 } }
\cs_generate_variant:Nn \__draw_point_interpolate_distance:nnnnn { e }
```

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

\draw_point_interpolate_arcaxes:nnnnnn
aw_point_interpolate_arcaxes_auxi:nnnnnnnnn
w_point_interpolate_arcaxes_auxii:nnnnnnnnn
w_point_interpolate_arcaxes_auxii:ennnnnnnn
aw_point_interpolate_arcaxes_auxiii:nnnnnnn
aw_point_interpolate_arcaxes_auxiii:ennnnnn
aw_point_interpolate_arcaxes_auxiv:nnnnnnnn
aw_point_interpolate_arcaxes_auxiv:eennnnnn

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.

```
\cs_new:Npn \draw_point_interpolate_arcaxes:nnnnnn #1#2#3#4#5#6
  {
    \__draw_point_process:nnnn
      { \__draw_point_interpolate_arcaxes_auxi:nnnnnnnnn {#1} {#5} {#6} }
      {#2} {#3} {#4}
  }
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxi:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_interpolate_arcaxes_auxii:ennnnnnnn
      { \fp_eval:n {#1} } {#2} {#3} {#4} {#5} {#6} {#7} {#8} {#9}
  }
```

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:ennnnnn
      { \fp_eval:n { #1 * (#3) + ( 1 - #1 ) * (#2) } }
      {#4} {#5} {#6} {#7} {#8} {#9}
  }
\cs_generate_variant:Nn \__draw_point_interpolate_arcaxes_auxii:nnnnnnnnn { e }
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxiii:nnnnnnn #1#2#3#4#5#6#7
  {
    \__draw_point_interpolate_arcaxes_auxiv:eennnnnn
      { \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 { e }
\cs_new:Npn \__draw_point_interpolate_arcaxes_auxiv:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \draw_point:n
      { #3 + #1 * #5 + #2 * #7 , #4 + #1 * #6 + #2 * #8 }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_arcaxes_auxiv:nnnnnnnn { ee }
```

(*End of definition for* `\draw_point_interpolate_arcaxes:nnnnnn` *and others. This function is documented on page* **??**.)

\draw_point_interpolate_curve:nnnnn
draw_point_interpolate_curve_auxi:nnnnnnnnn
raw_point_interpolate_curve_auxii:nnnnnnnnn
raw_point_interpolate_curve_auxii:ennnnnnnn
\draw_point_interpolate_curve_auxiii:nnnnnn
\draw_point_interpolate_curve_auxiii:ennnnn
\draw_point_interpolate_curve_auxiv:nnnnnn
\draw_point_interpolate_curve_auxv:nnw
\draw_point_interpolate_curve_auxv:eew
\draw_point_interpolate_curve_auxvi:n
raw_point_interpolate_curve_auxvii:nnnnnnnn
draw_point_interpolate_curve_auxviii:nnnnnn
draw_point_interpolate_curve_auxviii:eennnn

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:nnnnn
      { \__draw_point_interpolate_curve_auxi:nnnnnnnnn {#1} }
      {#2} {#3} {#4} {#5}
  }
\cs_new:Npn \__draw_point_interpolate_curve_auxi:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__draw_point_interpolate_curve_auxii:ennnnnnnn
      { \fp_eval:n {#1} }
      {#2} {#3} {#4} {#5} {#6} {#7} {#8} {#9}
  }
```

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:ennnnn
      { \fp_eval:n { 1 - #1 } }
      {#1}
      { {#2} {#3} } { {#4} {#5} } { {#6} {#7} } { {#8} {#9} }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_curve_auxii:nnnnnnnnn { e }
%    \begin{macrocode}
%   We need to do the first cycle, but haven't got enough arguments to keep
%   everything in play at once. So here we use a bit 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 { e }
\cs_new:Npn \__draw_point_interpolate_curve_auxiv:nnnnnn #1#2#3#4#5#6
  {
    \__draw_point_interpolate_curve_auxv:eew
      { \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 { ee }
%    \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:eeeenn
      { \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:n
      { #5 * #3 + #6 * #1 , #5 * #4 + #6 * #2 }
  }
\cs_generate_variant:Nn \__draw_point_interpolate_curve_auxviii:nnnnnn { eeee }
```

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

5.7 Vector support

As well as co-ordinates relative to the drawing

\l__draw_xvec_x_dim
\l__draw_xvec_y_dim
\l__draw_yvec_x_dim
\l__draw_yvec_y_dim
\l__draw_zvec_x_dim
\l__draw_zvec_y_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 of definition for* `\l__draw_xvec_x_dim` *and others.*)

\draw_xvec:n
\draw_yvec:n
\draw_zvec:n
__draw_vec:nn
__draw_vec:nnn

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 of 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`
`\__draw_point_vec:nn`
`\__draw_point_vec:ee`
`\draw_point_vec:nnn`
`\__draw_point_vec:nnn`
`\__draw_point_vec:eee`

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:ee { \fp_eval:n {#1} } { \fp_eval:n {#2} } }
\cs_new:Npn \__draw_point_vec:nn #1#2
  {
    \draw_point: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 { ee }
\cs_new:Npn \draw_point_vec:nnn #1#2#3
  {
    \__draw_point_vec:eee
      { \fp_eval:n {#1} } { \fp_eval:n {#2} } { \fp_eval:n {#3} }
  }
\cs_new:Npn \__draw_point_vec:nnn #1#2#3
  {
    \draw_point: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 { eee }
```

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

`\draw_point_vec_polar:nn`
`\draw_point_vec_polar:nnn`
`\__draw_point_vec_polar:nnn`
`\__draw_point_vec_polar:enn`

Much the same as the core polar approach.

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

```
  { \draw_point_vec_polar:nnn {#1} {#1} {#2} }
\cs_new:Npn \draw_point_vec_polar:nnn #1#2#3
  { \__draw_draw_vec_polar:enn { \fp_eval:n {#3} } {#1} {#2} }
\cs_new:Npn \__draw_draw_vec_polar:nnn #1#2#3
  {
    \draw_point: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 { e }
```

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

5.8 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_matrix_active_bool
      {
        \draw_point:n
          {
            (
                \l__draw_matrix_a_fp * #1
              + \l__draw_matrix_c_fp * #2
              + \l__draw_xshift_dim
            )
            ,
            (
                \l__draw_matrix_b_fp * #1
              + \l__draw_matrix_d_fp * #2
              + \l__draw_yshift_dim
            )
          }
      }
      {
        \draw_point:n
          {
              (#1, #2)
            + ( \l__draw_xshift_dim , \l__draw_yshift_dim )
          }
      }
  }
```

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

__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_matrix_active_bool
      {
        \draw_point:n
          {
            (
                \l__draw_matrix_a_fp * #1
              + \l__draw_matrix_c_fp * #2
            )
            ,
            (
                \l__draw_matrix_b_fp * #1
              + \l__draw_matrix_d_fp * #2
            )
          }
      }
      { \draw_point:n { (#1, #2) } }
  }
```

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

```
⟨/package⟩
```

6 l3draw-scopes implementation

```
⟨*package⟩
⟨@@=draw⟩
```

This sub-module covers more-or-less the same ideas as `pgfcorescopes.code.tex`. At present, equivalents of the following are currently absent:

- `\pgftext`: This is covered at this level by the coffin-based interface `\draw_coffin_use:Nnn`

6.1 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 of definition for* `\g__draw_xmax_dim` *and others.*)

\l_draw_bb_update_bool

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

```
\bool_new:N \l_draw_bb_update_bool
```

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

`\l__draw_layer_main_box` Box for setting the drawing itself and the top-level layer.

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

(*End of definition for* `\l__draw_layer_main_box`.)

`\g__draw_id_int` The drawing number.

```
\int_new:N \g__draw_id_int
```

(*End of definition for* `\g__draw_id_int`.)

`\__draw_reset_bb:` A simple auxiliary.

```
\cs_new_protected:Npn \__draw_reset_bb:
  {
    \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 }
  }
```

(*End of definition for* `\__draw_reset_bb:`.)

`\draw_begin:`
`\draw_end:`

Drawings are created by setting them into a box, then adjusting the box before inserting into the surroundings. Color is set here using the drawing mechanism largely as it then sets up the internal data structures. It may be that a coffin construct is better here in the longer term: that may become clearer as the code is completed. As we need to avoid any insertion of baseline skips, the outer box here has to be an `hbox`. To allow for layers, there is some box nesting: notice that we

```
\cs_new_protected:Npn \draw_begin:
  {
    \group_begin:
      \int_gincr:N \g__draw_id_int
      \hbox_set:Nw \l__draw_main_box
        \__draw_backend_begin:
        \__draw_reset_bb:
        \__draw_path_reset_limits:
        \bool_set_true:N \l_draw_bb_update_bool
        \draw_transform_matrix_reset:
        \draw_transform_shift_reset:
        \__draw_softpath_clear:
        \draw_linewidth:n { \l_draw_default_linewidth_dim }
        \color_select:n { . }
        \draw_nonzero_rule:
        \draw_cap_butt:
        \draw_join_miter:
        \draw_miterlimit:n { 10 }
        \draw_dash_pattern:nn { } { 0cm }
        \hbox_set:Nw \l__draw_layer_main_box
  }
\cs_new_protected:Npn \draw_end:
  {
          \__draw_baseline_finalise:w
```

```
          \exp_args:NNNV \hbox_set_end:
          \clist_set:Nn \l_draw_layers_clist \l_draw_layers_clist
          \__draw_layers_insert:
        \__draw_backend_end:
      \hbox_set_end:
      \dim_compare:nNnT \g__draw_xmin_dim = \c_max_dim
        {
          \dim_gzero:N \g__draw_xmax_dim
          \dim_gzero:N \g__draw_xmin_dim
          \dim_gzero:N \g__draw_ymax_dim
          \dim_gzero:N \g__draw_ymin_dim
        }
      \__draw_finalise:
      \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
    \group_end:
  }
```

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

`\__draw_finalise:`
`\__draw_finalise_baseline:n`

Finalising the (vertical) size of the output depends on whether we have an explicit baseline or not. To allow for that, we have two functions, and the one that's used depends on whether the user has set a baseline. Notice that in contrast to pgf we *do* allow for a non-zero depth if the explicit baseline is above the lowest edge of the initial bounding box.

```
\cs_new_protected:Npn \__draw_finalise:
  {
    \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_dp:Nn \l__draw_main_box { 0pt }
    \box_set_ht:Nn \l__draw_main_box
      { \g__draw_ymax_dim - \g__draw_ymin_dim }
  }
\cs_new_protected:Npn \__draw_finalise_baseline:n #1
  {
    \hbox_set:Nn \l__draw_main_box
      {
        \skip_horizontal:n { -\g__draw_xmin_dim }
        \box_move_down:nn
          {#1}
          { \box_use_drop:N \l__draw_main_box }
      }
    \box_set_dp:Nn \l__draw_main_box
      {
        \dim_max:nn
          { #1 - \g__draw_ymin_dim }
          { 0pt }
```

```
    }
  \box_set_ht:Nn \l__draw_main_box
    { \g__draw_ymax_dim - #1 }
}
```

(*End of definition for* `\__draw_finalise:` *and* `\__draw_finalise_baseline:n`.)

6.2 Baseline position

`\l__draw_baseline_bool`
`\l__draw_baseline_dim`

For tracking the explicit baseline and whether it is active.

```
\bool_new:N \l__draw_baseline_bool
\dim_new:N \l__draw_baseline_dim
```

(*End of definition for* `\l__draw_baseline_bool` *and* `\l__draw_baseline_dim`.)

`\draw_baseline:n`

A simple setting of the baseline along with the flag we need to know that it is active.

```
\cs_new_protected:Npn \draw_baseline:n #1
  {
    \bool_set_true:N \l__draw_baseline_bool
    \dim_set:Nn \l__draw_baseline_dim { \fp_to_dim:n {#1} }
  }
```

(*End of definition for* `\draw_baseline:n`. *This function is documented on page* **??**.)

`\__draw_baseline_finalise:w`

Rather than use a global data structure, we can arrange to put the baseline value at the right group level with a small amount of shuffling. That happens here.

```
\cs_new_protected:Npn \__draw_baseline_finalise:w #1 \__draw_finalise:
  {
    \bool_if:NTF \l__draw_baseline_bool
      {
        \use:e
          {
            \exp_not:n {#1}
            \__draw_finalise_baseline:n { \dim_use:N \l__draw_baseline_dim }
          }
      }
      { #1 \__draw_finalise: }
  }
```

(*End of definition for* `\__draw_baseline_finalise:w`.)

6.3 Scopes

`\l__draw_linewidth_dim`
`\l__draw_fill_color_tl`
`\l__draw_stroke_color_tl`

Storage for local variables.

```
\dim_new:N \l__draw_linewidth_dim
\tl_new:N \l__draw_fill_color_tl
\tl_new:N \l__draw_stroke_color_tl
```

(*End of definition for* `\l__draw_linewidth_dim`, `\l__draw_fill_color_tl`, *and* `\l__draw_stroke_color_tl`.)

\draw_scope_begin:
\draw_scope_begin:

As well as the graphics (and TEX) scope, also deal with global data structures.

```
\cs_new_protected:Npn \draw_scope_begin:
  {
    \__draw_backend_scope_begin:
    \group_begin:
      \dim_set_eq:NN \l__draw_linewidth_dim \g__draw_linewidth_dim
      \draw_path_scope_begin:
  }
\cs_new_protected:Npn \draw_scope_end:
  {
      \draw_path_scope_end:
      \dim_gset_eq:NN \g__draw_linewidth_dim \l__draw_linewidth_dim
    \group_end:
    \__draw_backend_scope_end:
  }
```

(*End of definition for* `\draw_scope_begin:`*. This function is documented on page* **??**.)

\l__draw_xmax_dim
\l__draw_xmin_dim
\l__draw_ymax_dim
\l__draw_ymin_dim

Storage for the bounding box.

```
\dim_new:N \l__draw_xmax_dim
\dim_new:N \l__draw_xmin_dim
\dim_new:N \l__draw_ymax_dim
\dim_new:N \l__draw_ymin_dim
```

(*End of definition for* `\l__draw_xmax_dim` *and others.*)

__draw_scope_bb_begin:
__draw_scope_bb_end:

The bounding box is simple: a straight group-based save and restore approach.

```
\cs_new_protected:Npn \__draw_scope_bb_begin:
  {
    \group_begin:
      \dim_set_eq:NN \l__draw_xmax_dim \g__draw_xmax_dim
      \dim_set_eq:NN \l__draw_xmin_dim \g__draw_xmin_dim
      \dim_set_eq:NN \l__draw_ymax_dim \g__draw_ymax_dim
      \dim_set_eq:NN \l__draw_ymin_dim \g__draw_ymin_dim
      \__draw_reset_bb:
  }
\cs_new_protected:Npn \__draw_scope_bb_end:
  {
      \dim_gset_eq:NN \g__draw_xmax_dim \l__draw_xmax_dim
      \dim_gset_eq:NN \g__draw_xmin_dim \l__draw_xmin_dim
      \dim_gset_eq:NN \g__draw_ymax_dim \l__draw_ymax_dim
      \dim_gset_eq:NN \g__draw_ymin_dim \l__draw_ymin_dim
    \group_end:
  }
```

(*End of definition for* `\__draw_scope_bb_begin:` *and* `\__draw_scope_bb_end:`*.*)

\draw_suspend_begin:
\draw_suspend_end:

Suspend all parts of a drawing.

```
\cs_new_protected:Npn \draw_suspend_begin:
  {
    \__draw_scope_bb_begin:
    \draw_path_scope_begin:
    \draw_transform_matrix_reset:
    \draw_transform_shift_reset:
```

```
    \__draw_layers_save:
  }
\cs_new_protected:Npn \draw_suspend_end:
  {
    \__draw_layers_restore:
    \draw_path_scope_end:
    \__draw_scope_bb_end:
  }
```

(*End of definition for* `\draw_suspend_begin:` *and* `\draw_suspend_end:`*. These functions are documented on page* **??**.)

```
⟨/package⟩
```

7 l3draw-softpath implementation

```
⟨*package⟩
⟨@@=draw⟩
```

7.1 Managing soft paths

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 so we use `\tl_build_...` functions.

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.

```
\tl_new:N \g__draw_softpath_main_tl
```

(*End of definition for* `\g__draw_softpath_main_tl`.)

`\l__draw_softpath_tmp_tl` Scratch space.

```
\tl_new:N \l__draw_softpath_tmp_tl
```

(*End of definition for* `\l__draw_softpath_tmp_tl`.)

`\g__draw_softpath_corners_bool` Allow for optimised path use.

```
\bool_new:N \g__draw_softpath_corners_bool
```

(*End of definition for* `\g__draw_softpath_corners_bool`.)

`\__draw_softpath_add:n`
`\__draw_softpath_add:o`
`\__draw_softpath_add:e`

```
\cs_new_protected:Npn \__draw_softpath_add:n
  { \tl_build_gput_right:Nn \g__draw_softpath_main_tl }
\cs_generate_variant:Nn \__draw_softpath_add:n { o, e }
```

(*End of definition for* `\__draw_softpath_add:n`.)

__draw_softpath_use:
__draw_softpath_clear:

Using and clearing is trivial.

```
\cs_new_protected:Npn \__draw_softpath_use:
  {
    \tl_build_get_intermediate:NN
      \g__draw_softpath_main_tl
      \l__draw_softpath_tmp_tl
    \l__draw_softpath_tmp_tl
  }
\cs_new_protected:Npn \__draw_softpath_clear:
  {
    \tl_build_gbegin:N \g__draw_softpath_main_tl
    \bool_gset_false:N \g__draw_softpath_corners_bool
  }
```

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

__draw_softpath_save:
__draw_softpath_restore:

Abstracted ideas to keep variables inside this submodule.

```
\cs_new_protected:Npn \__draw_softpath_save:
  {
    \tl_build_gend:N \g__draw_softpath_main_tl
    \tl_set_eq:NN
      \l__draw_softpath_main_tl
      \g__draw_softpath_main_tl
    \bool_set_eq:NN
      \l__draw_softpath_corners_bool
      \g__draw_softpath_corners_bool
    \__draw_softpath_clear:
  }
\cs_new_protected:Npn \__draw_softpath_restore:
  {
    \__draw_softpath_clear:
    \__draw_softpath_add:o \l__draw_softpath_main_tl
    \bool_gset_eq:NN
      \g__draw_softpath_corners_bool
      \l__draw_softpath_corners_bool
  }
```

(*End of definition for* `\__draw_softpath_save:` *and* `\__draw_softpath_restore:`.)

\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 of 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 of definition for* `\g__draw_softpath_move_bool`.)

__draw_softpath_closepath:
__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:e
      {
        \__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} }
    \bool_gset_true:N \g__draw_softpath_corners_bool
  }
\cs_generate_variant:Nn \__draw_softpath_roundpoint:nn { VV }
```

(*End of definition for* `\__draw_softpath_closepath:` *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. The support tokens for curves have to be different in meaning to a round point, hence being quark-like.

```
\cs_new_protected:Npn \__draw_softpath_close_op:nn #1#2
  { \__draw_backend_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
  { \__draw_backend_curveto:nnnnnn {#1} {#2} {#4} {#5} {#7} {#8} }
\cs_new_protected:Npn \__draw_softpath_curveto_opii:nn #1#2
  { \__draw_softpath_curveto_opii:nn }
\cs_new_protected:Npn \__draw_softpath_curveto_opiii:nn #1#2
  { \__draw_softpath_curveto_opiii:nn }
\cs_new_protected:Npn \__draw_softpath_lineto_op:nn #1#2
  { \__draw_backend_lineto:nn {#1} {#2} }
\cs_new_protected:Npn \__draw_softpath_moveto_op:nn #1#2
  { \__draw_backend_moveto:nn {#1} {#2} }
\cs_new_protected:Npn \__draw_softpath_roundpoint_op:nn #1#2
  { \__draw_softpath_roundpoint_op:nn }
\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
  { \__draw_backend_rectangle:nnnn {#1} {#2} {#4} {#5} }
\cs_new_protected:Npn \__draw_softpath_rectangle_opii:nn #1#2
  { \__draw_softpath_rectangle_opii:nn }
```

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

7.2 Rounding soft path corners

The aim here is to find corner rounding points and to replace them with arcs of appropriate length. The approach is exactly that in pgf: step through, find the corners, find the supporting data, do the rounding.

`\l__draw_softpath_main_tl`

For constructing the updated path.

```
\tl_new:N \l__draw_softpath_main_tl
```

(*End of definition for* `\l__draw_softpath_main_tl`.)

`\l__draw_softpath_part_tl`

Data structures.

```
\tl_new:N \l__draw_softpath_part_tl
\tl_new:N \l__draw_softpath_curve_end_tl
```

(*End of definition for* `\l__draw_softpath_part_tl`.)

`\l__draw_softpath_lastx_fp`
`\l__draw_softpath_lasty_fp`
`\l__draw_softpath_corneri_dim`
`\l__draw_softpath_cornerii_dim`
`\l__draw_softpath_first_tl`
`\l__draw_softpath_move_tl`

Position tracking: the token list data may be entirely empty or set to a co-ordinate.

```
\fp_new:N \l__draw_softpath_lastx_fp
\fp_new:N \l__draw_softpath_lasty_fp
\dim_new:N \l__draw_softpath_corneri_dim
\dim_new:N \l__draw_softpath_cornerii_dim
\tl_new:N \l__draw_softpath_first_tl
\tl_new:N \l__draw_softpath_move_tl
```

(*End of definition for* `\l__draw_softpath_lastx_fp` *and others.*)

\c__draw_softpath_arc_fp

The magic constant.

```
\fp_const:Nn \c__draw_softpath_arc_fp { 4/3 * (sqrt(2) - 1) }
```

(*End of definition for* \c__draw_softpath_arc_fp.)

__draw_softpath_round_corners:
__draw_softpath_round_loop:Nnn
__draw_softpath_round_action:nn
__draw_softpath_round_action:Nnn
__draw_softpath_round_action_curveto:NnnNnn
__draw_softpath_round_action_close:
__draw_softpath_round_lookahead:NnnNnn
__draw_softpath_round_roundpoint:NnnNnnNnn
__draw_softpath_round_calc:NnnNnn
__draw_softpath_round_calc:nnnnnn
__draw_softpath_round_calc:eVnnnn
__draw_softpath_round_calc:nnnnw
__draw_softpath_round_close:nn
__draw_softpath_round_close:w
__draw_softpath_round_end:

Rounding corners on a path means going through the entire path and adjusting it. As such, we avoid this entirely if we know there are no corners to deal with. Assuming there is work to do, we recover the existing path and start a loop.

```
\cs_new_protected:Npn \__draw_softpath_round_corners:
  {
    \bool_if:NT \g__draw_softpath_corners_bool
      {
        \group_begin:
          \tl_clear:N \l__draw_softpath_main_tl
          \tl_clear:N \l__draw_softpath_part_tl
          \fp_zero:N \l__draw_softpath_lastx_fp
          \fp_zero:N \l__draw_softpath_lasty_fp
          \tl_clear:N \l__draw_softpath_first_tl
          \tl_clear:N \l__draw_softpath_move_tl
          \tl_build_gend:N \g__draw_softpath_main_tl
          \exp_after:wN \__draw_softpath_round_loop:Nnn
            \g__draw_softpath_main_tl
            \q__draw_recursion_tail ? ?
            \q__draw_recursion_stop
        \group_end:
      }
    \bool_gset_false:N \g__draw_softpath_corners_bool
  }
```

The loop can take advantage of the fact that all soft path operations are made up of a token followed by two arguments. At this stage, there is a simple split: have we round a round point. If so, is there any actual rounding to be done: if the arcs have come through zero, just ignore it. In cases where we are not at a corner, we simply move along the path, allowing for any new part starting due to a `moveto`.

```
\cs_new_protected:Npn \__draw_softpath_round_loop:Nnn #1#2#3
  {
    \__draw_if_recursion_tail_stop_do:Nn #1 { \__draw_softpath_round_end: }
    \token_if_eq_meaning:NNTF #1 \__draw_softpath_roundpoint_op:nn
      { \__draw_softpath_round_action:nn {#2} {#3} }
      {
        \tl_if_empty:NT \l__draw_softpath_first_tl
          { \tl_set:Nn \l__draw_softpath_first_tl { {#2} {#3} } }
        \fp_set:Nn \l__draw_softpath_lastx_fp {#2}
        \fp_set:Nn \l__draw_softpath_lasty_fp {#3}
        \token_if_eq_meaning:NNTF #1 \__draw_softpath_moveto_op:nn
          {
            \tl_put_right:No \l__draw_softpath_main_tl
              \l__draw_softpath_move_tl
            \tl_put_right:No \l__draw_softpath_main_tl
              \l__draw_softpath_part_tl
            \tl_set:Nn \l__draw_softpath_move_tl { #1 {#2} {#3} }
            \tl_clear:N \l__draw_softpath_first_tl
            \tl_clear:N \l__draw_softpath_part_tl
          }
```

```
          { \tl_put_right:Nn \l__draw_softpath_part_tl { #1 {#2} {#3} } }
        \__draw_softpath_round_loop:Nnn
      }
  }
\cs_new_protected:Npn \__draw_softpath_round_action:nn #1#2
  {
    \dim_set:Nn \l__draw_softpath_corneri_dim {#1}
    \dim_set:Nn \l__draw_softpath_cornerii_dim {#2}
    \bool_lazy_and:nnTF
      { \dim_compare_p:nNn \l__draw_softpath_corneri_dim = { 0pt } }
      { \dim_compare_p:nNn \l__draw_softpath_cornerii_dim = { 0pt } }
      { \__draw_softpath_round_loop:Nnn }
      { \__draw_softpath_round_action:Nnn }
  }
```

We now have a round point to work on and have grabbed the next item in the path. There are only a few cases where we have to do anything. Each of them is picked up by looking for the appropriate action.

```
\cs_new_protected:Npn \__draw_softpath_round_action:Nnn #1#2#3
  {
    \tl_if_empty:NT \l__draw_softpath_first_tl
      { \tl_set:Nn \l__draw_softpath_first_tl { {#2} {#3} } }
    \token_if_eq_meaning:NNTF #1 \__draw_softpath_curveto_opi:nn
      { \__draw_softpath_round_action_curveto:NnnNnn }
      {
        \token_if_eq_meaning:NNTF #1 \__draw_softpath_close_op:nn
          { \__draw_softpath_round_action_close: }
          {
            \token_if_eq_meaning:NNTF #1 \__draw_softpath_lineto_op:nn
              { \__draw_softpath_round_lookahead:NnnNnn }
              { \__draw_softpath_round_loop:Nnn }
          }
      }
      #1 {#2} {#3}
  }
```

For a curve, we collect the two control points then move on to grab the end point and add the curve there: the second control point becomes our starter.

```
\cs_new_protected:Npn \__draw_softpath_round_action_curveto:NnnNnn
  #1#2#3#4#5#6
  {
    \tl_put_right:Nn \l__draw_softpath_part_tl
      { #1 {#2} {#3} #4 {#5} {#6} }
    \fp_set:Nn \l__draw_softpath_lastx_fp {#5}
    \fp_set:Nn \l__draw_softpath_lasty_fp {#6}
    \__draw_softpath_round_lookahead:NnnNnn
  }
\cs_new_protected:Npn \__draw_softpath_round_action_close:
  {
    \bool_lazy_and:nnTF
      { ! \tl_if_empty_p:N \l__draw_softpath_first_tl }
      { ! \tl_if_empty_p:N \l__draw_softpath_move_tl }
      {
        \exp_after:wN \__draw_softpath_round_close:nn
          \l__draw_softpath_first_tl
```

```
      }
      { \__draw_softpath_round_loop:Nnn }
  }
```

At this stage we have a current (sub)operation (`#1`) and the next operation (`#4`), and can therefore decide whether to round or not. In the case of yet another rounding marker, we have to look a bit further ahead.

```
\cs_new_protected:Npn \__draw_softpath_round_lookahead:NnnNnn #1#2#3#4#5#6
  {
    \bool_lazy_any:nTF
      {
        { \token_if_eq_meaning_p:NN #4 \__draw_softpath_lineto_op:nn }
        { \token_if_eq_meaning_p:NN #4 \__draw_softpath_curveto_opi:nn }
        { \token_if_eq_meaning_p:NN #4 \__draw_softpath_close_op:nn }
      }
      {
        \__draw_softpath_round_calc:NnnNnn
          \__draw_softpath_round_loop:Nnn
          {#5} {#6}
      }
      {
        \token_if_eq_meaning:NNTF #4 \__draw_softpath_roundpoint_op:nn
          { \__draw_softpath_round_roundpoint:NnnNnnNnn }
          { \__draw_softpath_round_loop:Nnn }
      }
    #1 {#2} {#3}
    #4 {#5} {#6}
  }
\cs_new_protected:Npn \__draw_softpath_round_roundpoint:NnnNnnNnn
  #1#2#3#4#5#6#7#8#9
  {
    \__draw_softpath_round_calc:NnnNnn
      \__draw_softpath_round_loop:Nnn
      {#8} {#9}
      #1 {#2} {#3}
    #4 {#5} {#6} #7 {#8} {#9}
  }
```

We now have all of the data needed to construct a rounded corner: all that is left to do is to work out the detail! At this stage, we have details of where the corner itself is (`#5`, `#6`), and where the next point is (`#2`, `#3`). There are two types of calculations to do. First, we need to interpolate from those two points in the direction of the corner, in order to work out where the curve we are adding will start and end. From those, plus the points we already have, we work out where the control points will lie. All of this is done in an expansion to avoid multiple calls to `\tl_put_right:Ne`. The end point of the line is worked out up-front and saved: we need that if dealing with a close-path operation.

```
\cs_new_protected:Npn \__draw_softpath_round_calc:NnnNnn #1#2#3#4#5#6
  {
    \tl_set:Ne \l__draw_softpath_curve_end_tl
      {
        \draw_point_interpolate_distance:nnn
          \l__draw_softpath_cornerii_dim
          { #5 , #6 } { #2 , #3 }
      }
```

```
    \tl_put_right:Ne \l__draw_softpath_part_tl
      {
        \exp_not:N #4
        \__draw_softpath_round_calc:eVnnnn
          {
            \draw_point_interpolate_distance:nnn
              \l__draw_softpath_corneri_dim
              { #5 , #6 }
              {
                \l__draw_softpath_lastx_fp ,
                \l__draw_softpath_lasty_fp
              }
          }
          \l__draw_softpath_curve_end_tl
          {#5} {#6} {#2} {#3}
      }
    \fp_set:Nn \l__draw_softpath_lastx_fp {#5}
    \fp_set:Nn \l__draw_softpath_lasty_fp {#6}
    #1
  }
```

At this stage we have the two curve end points, but they are in co-ordinate form. So we split them up (with some more reordering).

```
\cs_new:Npn \__draw_softpath_round_calc:nnnnnn #1#2#3#4#5#6
  {
    \__draw_softpath_round_calc:nnnnw {#3} {#4} {#5} {#6}
      #1 \s__draw_mark #2 \s__draw_stop
  }
\cs_generate_variant:Nn \__draw_softpath_round_calc:nnnnnn { eV }
```

The calculations themselves are relatively straight-forward, as we use a quadratic Bézier curve.

```
\cs_new:Npn \__draw_softpath_round_calc:nnnnw
  #1#2#3#4 #5 , #6 \s__draw_mark #7 , #8 \s__draw_stop
  {
    {#5} {#6}
    \exp_not:N \__draw_softpath_curveto_opi:nn
      {
        \fp_to_dim:n
          { #5 + \c__draw_softpath_arc_fp * ( #1 - #5 ) }
      }
      {
        \fp_to_dim:n
          { #6 + \c__draw_softpath_arc_fp * ( #2 - #6 ) }
      }
    \exp_not:N \__draw_softpath_curveto_opii:nn
      {
        \fp_to_dim:n
          { #7 + \c__draw_softpath_arc_fp * ( #1 - #7 ) }
      }
      {
        \fp_to_dim:n
          { #8 + \c__draw_softpath_arc_fp* ( #2 - #8 ) }
      }
    \exp_not:N \__draw_softpath_curveto_opiii:nn
```

```
    {#7} {#8}
  }
```

To deal with a close-path operation, we need to do some manipulation. It needs to be treated as a line operation for rounding, and then have the close path operation re-added at the point where the curve ends. That means saving the end point in the calculation step (see earlier), and shuffling a lot.

```
\cs_new_protected:Npn \__draw_softpath_round_close:nn #1#2
  {
    \use:e
      {
        \__draw_softpath_round_calc:NnnNnn
          {
            \tl_set:Ne \exp_not:N \l__draw_softpath_move_tl
              {
                \__draw_softpath_moveto_op:nn
                \exp_not:N \exp_after:wN
                  \exp_not:N \__draw_softpath_round_close:w
                  \exp_not:N \l__draw_softpath_curve_end_tl
                    \s__draw_stop
              }
            \use:e
              {
                \exp_not:N \exp_not:N \exp_not:N \use_i:nnnn
                  {
                    \__draw_softpath_round_loop:Nnn
                      \__draw_softpath_close_op:nn
                      \exp_not:N \exp_after:wN
                        \exp_not:N \__draw_softpath_round_close:w
                        \exp_not:N \l__draw_softpath_curve_end_tl
                          \s__draw_stop
                  }
              }
          }
          {#1} {#2}
          \__draw_softpath_lineto_op:nn
          \exp_after:wN \use_none:n \l__draw_softpath_move_tl
      }
  }
\cs_new:Npn \__draw_softpath_round_close:w #1 , #2 \s__draw_stop { {#1} {#2} }
```

Tidy up the parts of the path, complete the built token list and put it back into action.

```
\cs_new_protected:Npn \__draw_softpath_round_end:
  {
    \tl_put_right:No \l__draw_softpath_main_tl
      \l__draw_softpath_move_tl
    \tl_put_right:No \l__draw_softpath_main_tl
      \l__draw_softpath_part_tl
    \tl_build_gbegin:N \g__draw_softpath_main_tl
    \__draw_softpath_add:o \l__draw_softpath_main_tl
  }
```

(*End of definition for* `\__draw_softpath_round_corners:` *and others.*)

```
⟨/package⟩
```

8 l3draw-state implementation

```
⟨*package⟩
⟨@@=draw⟩
```

This sub-module covers more-or-less the same ideas as `pgfcoregraphicstate.code.tex`. At present, equivalents of the following are currently absent:

- `\pgfsetinnerlinewidth`, `\pgfinnerlinewidth`, `\pgfsetinnerstrokecolor`, `\pgfsetinnerstrok`
  Likely to be added on further work is done on paths/stroking.

`\g__draw_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
```

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

`\draw_linewidth:n` Set the linewidth: we need a wrapper as this has to pass to the driver layer.

```
\cs_new_protected:Npn \draw_linewidth:n #1
  {
    \dim_gset:Nn \g__draw_linewidth_dim { \fp_to_dim:n {#1} }
    \__draw_backend_linewidth:n \g__draw_linewidth_dim
  }
```

(*End of definition for* `\draw_linewidth:n`. *This function is documented on page* **??**.)

`\draw_dash_pattern:nn`
`\l__draw_tmp_seq` Evaluated all of the list and pass it to the driver layer.

```
\cs_new_protected:Npn \draw_dash_pattern:nn #1#2
  {
    \group_begin:
      \seq_set_from_clist:Nn \l__draw_tmp_seq {#1}
      \seq_set_map:NNn \l__draw_tmp_seq \l__draw_tmp_seq
        { \fp_to_dim:n {##1} }
      \use:e
        {
          \__draw_backend_dash_pattern:nn
            { \seq_use:Nn \l__draw_tmp_seq { , } }
            { \fp_to_dim:n {#2} }
        }
    \group_end:
  }
\seq_new:N \l__draw_tmp_seq
```

(*End of definition for* `\draw_dash_pattern:nn` *and* `\l__draw_tmp_seq`. *This function is documented on page* **??**.)

`\draw_miterlimit:n` Pass through to the driver layer.

```
\cs_new_protected:Npn \draw_miterlimit:n #1
  { \exp_args:Ne \__draw_backend_miterlimit:n { \fp_eval:n {#1} } }
```

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

`\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: { \__draw_backend_cap_butt: }
\cs_new_protected:Npn \draw_cap_rectangle: { \__draw_backend_cap_rectangle: }
\cs_new_protected:Npn \draw_cap_round: { \__draw_backend_cap_round: }
\cs_new_protected:Npn \draw_evenodd_rule: { \__draw_backend_evenodd_rule: }
\cs_new_protected:Npn \draw_nonzero_rule: { \__draw_backend_nonzero_rule: }
\cs_new_protected:Npn \draw_join_bevel: { \__draw_backend_join_bevel: }
\cs_new_protected:Npn \draw_join_miter: { \__draw_backend_join_miter: }
\cs_new_protected:Npn \draw_join_round: { \__draw_backend_join_round: }
```

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

```
⟨/package⟩
```

9 l3draw-transforms implementation

```
⟨*package⟩
⟨@@=draw⟩
```

This sub-module covers more-or-less the same ideas as `pgfcoretransformations.code.tex`. At present, equivalents of the following are currently absent:

- `\pgfgettransform`, `\pgfgettransformentries`: Awaiting use cases.
- `\pgftransformlineattime`, `\pgftransformarcaxesattime`, `\pgftransformcurveattime`: Need to look at the use cases for these to fully understand them.
- `\pgftransformarrow`: Likely to be done when other arrow functions are added.
- `\pgftransformationadjustments`: Used mainly by CircuiTi*k*Z although also for shapes, likely needs more use cases before addressing.
- `\pgflowlevelsynccm`, `\pgflowlevel`: Likely to be added when use cases are encountered in other parts of the code.
- `\pgfviewboxscope`: Seems very speicalied, need to understand the requirements here.

`\l__draw_matrix_active_bool`

An internal flag to avoid redundant calculations.

```
\bool_new:N \l__draw_matrix_active_bool
```

(*End of definition for* `\l__draw_matrix_active_bool`*.*)

`\l__draw_matrix_a_fp`
`\l__draw_matrix_b_fp`
`\l__draw_matrix_c_fp`
`\l__draw_xshift_dim`
`\l__draw_yshift_dim`

The active matrix and shifts.

```
\fp_new:N \l__draw_matrix_a_fp
\fp_new:N \l__draw_matrix_b_fp
\fp_new:N \l__draw_matrix_c_fp
\fp_new:N \l__draw_matrix_d_fp
\dim_new:N \l__draw_xshift_dim
\dim_new:N \l__draw_yshift_dim
```

(*End of definition for* `\l__draw_matrix_a_fp` *and others.*)

\draw_transform_matrix_reset:
\draw_transform_shift_reset:

Fast resetting.

```
\cs_new_protected:Npn \draw_transform_matrix_reset:
  {
    \fp_set:Nn \l__draw_matrix_a_fp { 1 }
    \fp_zero:N \l__draw_matrix_b_fp
    \fp_zero:N \l__draw_matrix_c_fp
    \fp_set:Nn \l__draw_matrix_d_fp { 1 }
    \bool_set_false:N \l__draw_matrix_active_bool
  }
\cs_new_protected:Npn \draw_transform_shift_reset:
  {
    \dim_zero:N \l__draw_xshift_dim
    \dim_zero:N \l__draw_yshift_dim
  }
\draw_transform_matrix_reset:
\draw_transform_shift_reset:
```

(*End of definition for* `\draw_transform_matrix_reset:` *and* `\draw_transform_shift_reset:`*. These functions are documented on page* **??**.)

\draw_transform_matrix_absolute:nnnn
\draw_transform_shift_absolute:n
__draw_transform_shift_absolute:nn

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.

```
\cs_new_protected:Npn \draw_transform_matrix_absolute:nnnn #1#2#3#4
  {
    \fp_set:Nn \l__draw_matrix_a_fp {#1}
    \fp_set:Nn \l__draw_matrix_b_fp {#2}
    \fp_set:Nn \l__draw_matrix_c_fp {#3}
    \fp_set:Nn \l__draw_matrix_d_fp {#4}
    \bool_lazy_all:nTF
      {
        { \fp_compare_p:nNn \l__draw_matrix_a_fp = \c_one_fp }
        { \fp_compare_p:nNn \l__draw_matrix_b_fp = \c_zero_fp }
        { \fp_compare_p:nNn \l__draw_matrix_c_fp = \c_zero_fp }
        { \fp_compare_p:nNn \l__draw_matrix_d_fp = \c_one_fp }
      }
      { \bool_set_false:N \l__draw_matrix_active_bool }
      { \bool_set_true:N \l__draw_matrix_active_bool }
  }
\cs_new_protected:Npn \draw_transform_shift_absolute:n #1
  {
    \__draw_point_process:nn
      { \__draw_transform_shift_absolute:nn } {#1}
  }
\cs_new_protected:Npn \__draw_transform_shift_absolute:nn #1#2
  { \__draw_transform_shift:nnnn { 0pt } { 0pt } {#1} {#2} }
```

(*End of definition for* `\draw_transform_matrix_absolute:nnnn`*,* `\draw_transform_shift_absolute:n`*, and* `\__draw_transform_shift_absolute:nn`*. These functions are documented on page* **??**.)

\draw_transform_matrix:nnnn
__draw_transform:nnnn
\draw_transform_shift:n
__draw_transform_shift:nn

Much the same story for adding to an existing matrix, with a bit of pre-expansion so that the calculation uses "frozen" values.

```
\cs_new_protected:Npn \draw_transform_matrix:nnnn #1#2#3#4
  {
    \use:e
```

```
      {
        \__draw_transform:nnnn
          { \fp_eval:n {#1} }
          { \fp_eval:n {#2} }
          { \fp_eval:n {#3} }
          { \fp_eval:n {#4} }
      }
  }
\cs_new_protected:Npn \__draw_transform:nnnn #1#2#3#4
  {
    \use:e
      {
        \draw_transform_matrix_absolute:nnnn
          { #1 * \l__draw_matrix_a_fp + #2 * \l__draw_matrix_c_fp }
          { #1 * \l__draw_matrix_b_fp + #2 * \l__draw_matrix_d_fp }
          { #3 * \l__draw_matrix_a_fp + #4 * \l__draw_matrix_c_fp }
          { #3 * \l__draw_matrix_b_fp + #4 * \l__draw_matrix_d_fp }
        }
  }
\cs_new_protected:Npn \draw_transform_shift:n #1
  {
    \__draw_point_process:nn
      { \__draw_transform_shift:nn } {#1}
  }
\cs_new_protected:Npn \__draw_transform_shift:nn #1#2
  {
    \__draw_transform_shift:nnnn
      \l__draw_xshift_dim
      \l__draw_yshift_dim
      {#1} {#2}
  }
```

(*End of definition for* `\draw_transform_matrix:nnnn` *and others. These functions are documented on page* **??**.)

`\__draw_transform_shift:nnnn`

Apply the current transformation matrix to the shift, then store the resulting values: we may or may not have a none-zero starting point here.

```
\cs_new_protected:Npn \__draw_transform_shift:nnnn #1#2#3#4
  {
    \dim_set:Nn \l__draw_xshift_dim
      {
        \fp_to_dim:n
          {
            #1 +
            ( #3 * \l__draw_matrix_a_fp + #4 * \l__draw_matrix_c_fp )
          }
      }
    \dim_set:Nn \l__draw_yshift_dim
      {
        \fp_to_dim:n
          {
            #2 +
            ( #3 * \l__draw_matrix_b_fp + #4 * \l__draw_matrix_d_fp )
          }
```

```
    }
  }
```

(*End of definition for* `\__draw_transform_shift:nnnn`*.*)

`\draw_transform_matrix_invert:`
`\__draw_transform_invert:n`
`\__draw_transform_invert:e`
`\draw_transform_shift_invert:`

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

```
\cs_new_protected:Npn \draw_transform_matrix_invert:
  {
    \bool_if:NT \l__draw_matrix_active_bool
      {
        \__draw_transform_invert:e
          {
            \fp_eval:n
              {
                1 /
                  (
                      \l__draw_matrix_a_fp * \l__draw_matrix_d_fp
                    - \l__draw_matrix_b_fp * \l__draw_matrix_c_fp
                  )
              }
          }
      }
  }
\cs_new_protected:Npn \__draw_transform_invert:n #1
  {
    \fp_set:Nn \l__draw_matrix_a_fp
      { \l__draw_matrix_d_fp * #1 }
    \fp_set:Nn \l__draw_matrix_b_fp
      { -\l__draw_matrix_b_fp * #1 }
    \fp_set:Nn \l__draw_matrix_c_fp
      { -\l__draw_matrix_c_fp * #1 }
    \fp_set:Nn \l__draw_matrix_d_fp
      { \l__draw_matrix_a_fp * #1 }
  }
\cs_generate_variant:Nn \__draw_transform_invert:n { e }
\cs_new_protected:Npn \draw_transform_shift_invert:
  {
    \dim_set:Nn \l__draw_xshift_dim { -\l__draw_xshift_dim }
    \dim_set:Nn \l__draw_yshift_dim { -\l__draw_yshift_dim }
  }
```

(*End of definition for* `\draw_transform_matrix_invert:`*,* `\__draw_transform_invert:n`*, and* `\draw_transform_shift_invert:`*. These functions are documented on page* **??***.*)

`\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_transform_triangle:nnnnnn }
          {#1}
      }
      {#2} {#3}
  }
```

```
\cs_new_protected:Npn \__draw_transform_triangle:nnnnnn #1#2#3#4#5#6
  {
    \use:e
      {
        \draw_transform_matrix_absolute:nnnn
          { #3 - #1 }
          { #4 - #2 }
          { #5 - #1 }
          { #6 - #2 }
        \draw_transform_shift_absolute:n { #1 , #2 }
      }
  }
```

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

`\draw_transform_scale:n`
`\draw_transform_xscale:n`
`\draw_transform_yscale:n`
`\draw_transform_xshift:n`
`\draw_transform_yshift:n`
`\draw_transform_xslant:n`
`\draw_transform_yslant:n`

Lots of shortcuts.

```
\cs_new_protected:Npn \draw_transform_scale:n #1
  { \draw_transform_matrix:nnnn { #1 } { 0 } { 0 } { #1 } }
\cs_new_protected:Npn \draw_transform_xscale:n #1
  { \draw_transform_matrix:nnnn { #1 } { 0 } { 0 } { 1 } }
\cs_new_protected:Npn \draw_transform_yscale:n #1
  { \draw_transform_matrix:nnnn { 1 } { 0 } { 0 } { #1 } }
\cs_new_protected:Npn \draw_transform_xshift:n #1
  { \draw_transform_shift:n { #1 , 0pt } }
\cs_new_protected:Npn \draw_transform_yshift:n #1
  { \draw_transform_shift:n { 0pt , #1 } }
\cs_new_protected:Npn \draw_transform_xslant:n #1
  { \draw_transform_matrix:nnnn { 1 } { 0 } { #1 } { 1 } }
\cs_new_protected:Npn \draw_transform_yslant:n #1
  { \draw_transform_matrix:nnnn { 1 } { #1 } { 0 } { 1 } }
```

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

`\draw_transform_rotate:n`
`\__draw_transform_rotate:n`
`\__draw_transform_rotate:e`
`\__draw_transform_rotate:nn`
`\__draw_transform_rotate:ee`

Slightly more involved: evaluate the angle only once, and the sine and cosine only once.

```
\cs_new_protected:Npn \draw_transform_rotate:n #1
  { \__draw_transform_rotate:e { \fp_eval:n {#1} } }
\cs_new_protected:Npn \__draw_transform_rotate:n #1
  {
    \__draw_transform_rotate:ee
      { \fp_eval:n { cosd(#1) } }
      { \fp_eval:n { sind(#1) } }
  }
\cs_generate_variant:Nn \__draw_transform_rotate:n { e }
\cs_new_protected:Npn \__draw_transform_rotate:nn #1#2
  { \draw_transform_matrix:nnnn {#1} {#2} { -#2 } { #1 } }
\cs_generate_variant:Nn \__draw_transform_rotate:nn { ee }
```

(*End of definition for* `\draw_transform_rotate:n`, `\__draw_transform_rotate:n`, *and* `\__draw_transform_rotate:nn`. *This function is documented on page* **??**.)

```
⟨/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.

E

F

G

H

I

K

M

P