The celtic package

Andrew Stacey
loopspace@mathforge.org

1.0 from 2014/05/23

1 Introduction

This is a TikZ library for drawing Celtic knot diagrams. For user documentation, see the `celtic.pdf` file.

2 Implementation

2.1 Initialisation

Load the LaTeX3 basics ...

```
\usepackage{expl3}
\usepackage{xparse}
```

... and enter the Realm of the 3rd LaTeX.

```
\ExplSyntaxOn
```

Wrapper around `\tikz@scan@one@point` for the `add=<coord>` key.

```
\cs_new_nopar:Npn \celtic_shift:n #1
{
  \use:c{tikz@scan@one@point}\pgftransformshift #1\relax
}
```

We need one or two variables ...

```
\int_new:N \l__celtic_max_steps_int
\int_new:N \l__celtic_int
\int_new:N \l__celtic_flip_int
\int_new:N \l__celtic_width_int
\int_new:N \l__celtic_height_int
\int_new:N \l__celtic_x
\int_new:N \l__celtic_y
\int_new:N \l__celtic_dx
\int_new:N \l__celtic_dy
\int_new:N \l__celtic_ox
\int_new:N \l__celtic_oy
\int_new:N \l__celtic_lout
\int_new:N \l__celtic_cross_int
```

```
\int_new:N \l__celtic_component_int
\fp_new:N \l__celtic_clip_fp
\fp_new:N \l__celtic_inner_clip_fp
\fp_new:N \l__celtic_inner_fp
\fp_new:N \l__celtic_outer_fp
\seq_new:N \l__celtic_path_seq
\seq_new:N \l__celtic_component_seq
\seq_new:N \l__celtic_crossing_seq
\seq_new:N \l__celtic_tmpa_seq
\clist_new:N \l__celtic_tmpa_clist
\tl_new:N \l__celtic_tmpa_tl
\tl_new:N \l__celtic_path_tl
\tl_new:N \g__celtic_colon_tl
\tl_new:N \l__celtic_bar_tl
\tl_new:N \l__celtic_active_bar_tl
\bool_new:N \l__celtic_bounce_bool
\bool_new:N \l__celtic_pbounce_bool
```

Define our warning message.

```
\msg_new:nnnn { celtic } { max~ steps } { Limit~ of~ number~ of~ steps~ exceeded~ \msg_line_co
{ Paths~ may~ not~ be~ correctly~ constructed.~
Consider~ raising~ the~ limit~ from  \int_use:N \l__celtic_max_steps_int.}
```

Using a colon for a range separator was possibly not the best idea I ever had, seeing as LaTeX3 alters its catcode. So we need to get creative.

```
\group_begin:
\char_set_lccode:nn {`;}{`:}
\tl_to_lowercase:n {
\group_end:
  \tl_set:Nn \g__celtic_colon_tl {;}
}
```

Some packages mess with the catcode of `|`.

```
\tl_set:Nn \l__celtic_bar_tl {|}
\group_begin:
\char_set_catcode_active:N \|
\tl_gset:Nn \l__celtic_active_bar_tl {|}
\group_end:
```

We need a few variants of standard LaTeX3 functions.

```
\cs_generate_variant:Nn \tl_if_single_p:N {c}
\cs_generate_variant:Nn \tl_if_single:NTF {cTF}
\cs_generate_variant:Nn \tl_if_eq:nnTF {xnTF}
\cs_generate_variant:Nn \tl_head:N {c}
\cs_generate_variant:Nn \tl_tail:N {c}
\cs_generate_variant:Nn \tl_if_eq:nnTF {vnTF}
\cs_generate_variant:Nn \tl_if_in:nnTF {nVTF}
```

Initialise a few variables.

```
\int_set:Nn \l__celtic_max_steps_int {20}
\fp_set:Nn \l__celtic_inner_fp {1}
\fp_set:Nn \l__celtic_outer_fp {2}
```

The following functions are for parsing and setting the crossing information.

`\celtic_do_crossing:nnn` This function sets the information for a particular crossing. The first argument can be empty, meaning "ignore this crossing as a starting point", or it should be one of `|` or `-` to denote the wall type that is placed at this crossing.

```
\cs_new_nopar:Npn \celtic_do_crossing:nnn #1#2#3
{
  \tl_if_empty:nTF {#1}
  {
    \tl_clear:c {crossing used \int_eval:n {#2} - \int_eval:n {#3}}
  }
  {
    \tl_set:cn {crossing \int_eval:n {#2} - \int_eval:n{#3}}{#1}
  }
}
```

`\celtic_maybe_symmetric:nnnn` If a crossing is designated as symmetric, we repeat the action four times. This macro tests to see if it is symmetric or not and acts accordingly.

```
\cs_new_nopar:Npn \celtic_maybe_symmetric:nnnn #1#2#3#4
{
  \tl_if_empty:nTF {#1}
  {
    \celtic_do_crossing:nnn {#2}{#3}{#4}
  }
  {
    \celtic_do_crossing:nnn {#2}{#3}{#4}
    \celtic_do_crossing:nnn {#2}{\l__celtic_width_int - #3}{#4}
    \celtic_do_crossing:nnn {#2}{#3}{\l__celtic_height_int - #4}
    \celtic_do_crossing:nnn {#2}{\l__celtic_width_int - #3}{\l__celtic_height_int - #4}
  }
}
```

`\celtic_maybe_xrange:nnnn` The x-coordinate might be a range. If it is, it contains a colon (with the normal catcode). So we test for a colon and act accordingly.

```
\cs_new_nopar:Npn \celtic_maybe_xrange:nnnn #1#2#3#4
{
  \tl_if_in:nVTF {#3} \g__celtic_colon_tl
  {
    \celtic_do_xrange:w {#1}{#2}#3\q_stop{#4}
  }
  {
    \celtic_maybe_yrange:nnnn {#1}{#2}{#3}{#4}
  }
}
```

`\celtic_maybe_yrange:nnnn` Same with the y-coordinate.

```
\cs_new_nopar:Npn \celtic_maybe_yrange:nnnn #1#2#3#4
{
  \tl_if_in:nVTF {#4} \g__celtic_colon_tl
```

```
  {
    \celtic_do_yrange:w {#1}{#2}{#3}#4\q_stop
  }
  {
    \celtic_maybe_symmetric:nnnn {#1}{#2}{#3}{#4}
  }
}
```

When processing ranges, we need to use colons with the original catcode. We've stored one in `\g__celtic_colon_tl` but we need to use it in actuality. So we make a token list containing the definitions we want to make, expanding `\g__celtic_colon_tl` to its colon, but not expanding anything else.

```
\tl_set:Nx \l_tmpa_tl
{
```

`\celtic_do_xrange:w` This splits the x-coordinate into a range and repeats the function for each intermediate value.

```
  \exp_not:N \cs_new_nopar:Npn \exp_not:N \celtic_do_xrange:w ##1##2##3\tl_use:N \g__celtic_co
  {
    \exp_not:N \int_step_inline:nnnn {##3} {2} {##4}
    {
      \exp_not:N \celtic_maybe_yrange:nnnn {##1}{##2} {####1}{##5}
    }
  }
```

`\celtic_do_yrange:w` Same, for the y-coordinate.

```
  \exp_not:N \cs_new_nopar:Npn \exp_not:N \celtic_do_yrange:w ##1##2##3##4\tl_use:N \g__celtic_
  {
    \exp_not:N \int_step_inline:nnnn {##4} {2} {##5}
    {
      \exp_not:N \celtic_maybe_symmetric:nnnn {##1}{##2}{##3}{####1}
    }
  }
}
```

Now we use the above token list to make our definitions with the right colon in them.

```
\tl_use:N \l_tmpa_tl
```

The next functions are those that take the individual crossing specifications from the key/value list and begin the process of converting the data to an action to be taken for a specific crossing.

`\celtic_ignore_crossings:w`

```
\cs_new_nopar:Npn \celtic_ignore_crossings:w #1,#2\q_stop
{
  \celtic_maybe_xrange:nnnn {}{}{#1}{#2}
}
```

\celtic_ignore_symmetric_crossings:w

```
\cs_new_nopar:Npn \celtic_ignore_symmetric_crossings:w #1,#2\q_stop
{
  \celtic_maybe_xrange:nnnn {s}{}{#1}{#2}
}
```

\celtic_set_crossings:w

```
\cs_new_nopar:Npn \celtic_set_crossings:w #1,#2,#3\q_stop
{
  \celtic_maybe_xrange:nnnn {}{#3}{#1}{#2}
}
```

\celtic_set_symmetric_crossings:w

```
\cs_new_nopar:Npn \celtic_set_symmetric_crossings:w #1,#2,#3\q_stop
{
  \celtic_maybe_xrange:nnnn {s}{#3}{#1}{#2}
}
```

\celtic_next_crossing: This is the function that does all the work. Starting from an undercrossing, it computes the segment leading to the next undercrossing working out all of the "bounces" on the way.

```
\cs_new_nopar:Npn \celtic_next_crossing:
{
```

Clear our starting conditions.

```
  \int_zero:N \l__celtic_cross_int
  \tl_clear:N \l__celtic_crossing_tl
  \tl_clear:N \l__celtic_path_tl
  \bool_set_false:N \l__celtic_bounce_tl
```

Start our path with a move to the initial point and record our current direction.

```
  \tl_put_right:Nx \l__celtic_path_tl {(\int_use:N \l__celtic_x, \int_use:N \l__celtic_y)}
  \int_set:Nn \l__celtic_lout {(90 - \l__celtic_dx * 45) * \l__celtic_dy}
```

We loop until we get to the second crossing on the path (the first will be the overpass).

```
  \bool_do_until:nn {\int_compare_p:n {\l__celtic_cross_int > 1}}
  {
```

We keep a record of whether the last bit contained a bounce.

```
    \bool_set_eq:NN \l__celtic_pbounce_bool \l__celtic_bounce_bool
    \bool_set_false:N \l__celtic_bounce_bool
```

Move to the next point in our current direction.

```
    \int_add:Nn \l__celtic_x {\l__celtic_dx}
    \int_add:Nn \l__celtic_y {\l__celtic_dy}
```

Now we look to see if we should bounce. Is the crossing defined?

```
    \tl_if_exist:cT {crossing \int_use:N \l__celtic_x - \int_use:N       \l__celtic_y}
    {
```

Yes, so we bounce. But which way?

```
      \tl_if_eq:cNTF {crossing \int_use:N \l__celtic_x - \int_use:N \l__celtic_y} \l__celtic_ba
      {
```

Vertical wall. Have we just bounced?

```
        \bool_if:NTF \l__celtic_pbounce_bool
        {
```

Yes, so the next part of the path is a right angle.

```
          \tl_put_right:Nn \l__celtic_path_tl { -| }
        }
        {
```

No, so the next part of the path is a curve. (This is where we use the direction that we recorded earlier.)

```
          \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_eval:n
{(90 - 45*\l__celtic_dx)*\l__celtic_dy}, in=\int_eval:n
{-90*\l__celtic_dy}] }
        }
```

We record the new direction and "bounce" our direction vector. Then we add our new point to the path (which, due to the bounce, is offset).

```
        \int_set:Nn \l__celtic_lout {90*\l__celtic_dy}
        \int_set:Nn \l__celtic_dx {-\l__celtic_dx}
        \tl_put_right:Nx  \l__celtic_path_tl {(\fp_eval:n {\int_use:N \l__celtic_x + .5 * \int
```

We bounced, so record that too.

```
        \bool_set_true:N \l__celtic_bounce_bool
      }
      {
```

At this point, we've bounced but our bounce was horizontal so we do the same as for the vertical but all turned round.

```
        \bool_if:NTF \l__celtic_pbounce_bool
        {
```

We're out from a bounce, so turn at right angles.

```
          \tl_put_right:Nn \l__celtic_path_tl { |- }
        }
        {
```

We're not out from a bounce, so we curve ...

```
          \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_eval:n {(90 - 45*\l__celtic_dx)*\l
        }
```

... and record our new direction and out angle.

```
        \int_set:Nn \l__celtic_lout {90-90*\l__celtic_dx}
        \int_set:Nn \l__celtic_dy {-\l__celtic_dy}
```

Now we add our new position (adjusted from the bounce) to the path.

```
        \tl_put_right:Nx \l__celtic_path_tl {(\int_use:N \l__celtic_x, \fp_eval:n {\int_use:N
```

And record the fact that we've bounced.

```
      \bool_set_true:N \l__celtic_bounce_bool
    }
  }
```

Now we check to see if we're at the edge of the rectangle, starting with the left.

```
  \int_compare:nT {\l__celtic_x == 0}
  {
```

Yes, so treat this as a vertical bounce.

```
    \bool_if:NTF \l__celtic_pbounce_bool
    {
```

Previous bounce, so right angle.

```
      \tl_put_right:Nn \l__celtic_path_tl { -| }
    }
    {
```

No previous bounce, so curve.

```
      \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_eval:n {(90 - 45*\l__celtic_dx)*\l__
    }
```

Record our out angle and change our direction.

```
    \int_set:Nn \l__celtic_lout {90*\l__celtic_dy}
    \int_set:Nn \l__celtic_dx {-\l__celtic_dx}
```

Add the correct position to the path.

```
    \tl_put_right:Nx  \l__celtic_path_tl {(\fp_eval:n {\int_use:N \l__celtic_x + .5 * \int_us
```

We've bounced.

```
    \bool_set_true:N \l__celtic_bounce_bool
  }
```

Same for the right-hand edge.

```
  \int_compare:nT {\l__celtic_x == \l__celtic_width_int}
  {
    \bool_if:NTF \l__celtic_pbounce_bool
    {
      \tl_put_right:Nn \l__celtic_path_tl { -| }
    }
    {
      \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_eval:n {(90 - 45*\l__celtic_dx)*\l__
    }
    \int_set:Nn \l__celtic_lout {90*\l__celtic_dy}
    \int_set:Nn \l__celtic_dx {-\l__celtic_dx}
    \tl_put_right:Nx \l__celtic_path_tl {(\fp_eval:n {\int_use:N \l__celtic_x + .5 * \int_use
    \bool_set_true:N \l__celtic_bounce_bool
  }
```

Now the lower edge.

```
    \int_compare:nT {\l__celtic_y == 0}
    {
      \bool_if:NTF \l__celtic_pbounce_bool
      {
        \tl_put_right:Nn \l__celtic_path_tl { |- }
      }
      {
        \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_eval:n {(90 - 45*\l__celtic_dx)*\l__
      }
      \int_set:Nn \l__celtic_lout {90-90*\l__celtic_dx}
      \int_set:Nn \l__celtic_dy {-\l__celtic_dy}
      \tl_put_right:Nx \l__celtic_path_tl {(\int_use:N \l__celtic_x, \fp_eval:n {\int_use:N \l
      \bool_set_true:N \l__celtic_bounce_bool
    }
```

And the upper edge.

```
    \int_compare:nT {\l__celtic_y == \l__celtic_height_int}
    {
      \bool_if:NTF \l__celtic_pbounce_bool
      {
        \tl_put_right:Nn \l__celtic_path_tl { |- }
      }
      {
        \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_eval:n {(90 - 45*\l__celtic_dx)*\l__
      }
      \int_set:Nn \l__celtic_lout {-90+90*\l__celtic_dx}
      \int_set:Nn \l__celtic_dy {-\l__celtic_dy}
      \tl_put_right:Nx \l__celtic_path_tl {(\int_use:N \l__celtic_x, \fp_eval:n {\int_use:N \l
      \bool_set_true:N \l__celtic_bounce_bool
    }
```

Did we bounce this time?

```
    \bool_if:NF \l__celtic_bounce_bool
    {
```

Did we bounce last time?

```
      \bool_if:NTF \l__celtic_pbounce_bool
      {
```

Yes, so the second half is a curve.

```
        \tl_put_right:Nx \l__celtic_path_tl { to[out=\int_use:N \l__celtic_lout,in=\int_eval:n
      }
      {
```

No, so the second half is a straight line.

```
        \tl_put_right:Nn \l__celtic_path_tl { -- }
      }
```

The next crossing.

```
      \tl_put_right:Nx  \l__celtic_path_tl { (\int_use:N         \l__celtic_x, \int_use:N \l__
```

If we haven't already gone over a crossing, this is our overcrossing.

```
      \tl_if_empty:NTF \l__celtic_crossing_tl
      {
```

So we record this as our overcrossing.

```
        \tl_set:Nx \l__celtic_crossing_tl {(\int_use:N         \l__celtic_x, \int_use:N \l__cel
      }
      {
```

Otherwise, it's the undercrossing so we note that we've visited this one.

```
        \tl_clear:c {crossing used \int_use:N \l__celtic_x - \int_use:N \l__celtic_y}
      }
```

Increment the crossing count.

```
      \int_incr:N \l__celtic_cross_int
```

Record our outward angle.

```
      \int_set:Nn \l__celtic_lout {(90 - \l__celtic_dx * 45) * \l__celtic_dy}
    }
  }
}
```

Now we set up the keys we'll use.

```
\keys_define:nn { celtic }
{
```

This sets the maximum number of steps in a path.

```
  max~ steps .int_set:N = \l__celtic_max_steps_int,
```

This flips the over/under crossings.

```
  flip .code:n = {
    \int_set:Nn \l__celtic_flip_int {-1}
  },
```

These set the size of the knot.

```
  width .int_set:N = \l__celtic_width_int,
  height .int_set:N = \l__celtic_height_int,
  size .code:n = {
```

The size is a CSV so we use a `clist` to separate it.

```
    \clist_set:Nn \l__celtic_tmpa_clist {#1}
    \clist_pop:NN \l__celtic_tmpa_clist \l__celtic_tmpa_tl
    \int_set:Nn \l__celtic_width_int {\l__celtic_tmpa_tl}
    \clist_pop:NN \l__celtic_tmpa_clist \l__celtic_tmpa_tl
    \int_set:Nn \l__celtic_height_int {\l__celtic_tmpa_tl}
  },
```

The size keys are placed in a separate group to make it possible to process them before all other keys.

```
  width  .groups:n = { size },
  height .groups:n = { size },
  size   .groups:n = { size },
```

The next keys set the various crossing behaviours.

```
  crossings .code:n = {
    \seq_set_split:Nnn \l__celtic_tmpa_seq {;} {#1}
    \seq_map_inline:Nn \l__celtic_tmpa_seq {
      \tl_if_empty:nF {##1}
      {
        \celtic_set_crossings:w ##1 \q_stop
      }
    }
  },
  symmetric~ crossings .code:n = {
    \seq_set_split:Nnn \l__celtic_tmpa_seq {;} {#1}
    \seq_map_inline:Nn \l__celtic_tmpa_seq {
      \tl_if_empty:nF {##1}
      {
        \celtic_set_symmetric_crossings:w ##1 \q_stop
      }
    }
  },
  ignore~ crossings .code:n ={
    \seq_set_split:Nnn \l__celtic_tmpa_seq {;} {#1}
    \seq_map_inline:Nn \l__celtic_tmpa_seq {
      \tl_if_empty:nF {##1}
      {
        \celtic_ignore_crossings:w ##1 \q_stop
      }
    }
  },
  ignore~ symmetric~ crossings .code:n ={
    \seq_set_split:Nnn \l__celtic_tmpa_seq {;} {#1}
    \seq_map_inline:Nn \l__celtic_tmpa_seq {
      \tl_if_empty:nF {##1}
      {
        \celtic_ignore_symmetric_crossings:w ##1 \q_stop
      }
    }
  },
```

The `style` key is passed on to `\tikzset`.

```
  style .code:n = {
    \tikzset {#1}
  },
```

This relocates the diagram.

```
  at .code:n = {
    \celtic_shift:n {#1}
  },
```

These set the margin for the clip regions.

```
  inner~ clip .fp_set:N = \l__celtic_inner_fp,
  outer~ clip .fp_set:N = \l__celtic_outer_fp,
}
```

`\CelticDrawPath` This is the user macro. Its mandatory argument is a list of key/value pairs.

```
\DeclareDocumentCommand \CelticDrawPath { m }
{
```

Get a nice clean initial state.

```
  \group_begin:
  \pgfscope
  \seq_clear:N \l__celtic_path_seq
  \seq_clear:N \l__celtic_component_seq
  \seq_clear:N \l__celtic_crossing_seq
  \int_set:Nn \l__celtic_flip_int {1}
```

Figure out if `|` is active or not (`fancyvrb` sets it active).

```
\int_compare:nT {\char_value_catcode:n {`\|} = 13}
{
  \tl_set_eq:NN \l__celtic_bar_tl \l__celtic_active_bar_tl
}
```

Clear all the crossing data.

```
  \int_step_inline:nnnn {1} {1} {\l__celtic_height_int-1}
  {
    \int_step_inline:nnnn {1 + \int_mod:nn {##1}{2}} {2} {\l__celtic_width_int-1}
{
  \tl_clear_new:c {crossing used ####1 - ##1}
  \tl_set:cn {crossing used ####1 - ##1} {X}
}
  }
```

Process the keys relating to the size of the knot.

```
  \keys_set_groups:nnn { celtic } { size } {#1}
```

Process all other keys.

```
  \keys_set_filter:nnn { celtic } { size } {#1}
```

Draw (maybe) the outer boundary.

```
  \path[celtic~ bar/.try, celtic~ surround/.try] (0,0) rectangle (\int_use:N \l__celtic_width_
```

Draw (maybe) the crossings.

```
  \int_step_inline:nnnn {1} {1} {\l__celtic_height_int-1}
  {
    \int_step_inline:nnnn {1 + \int_mod:nn {##1}{2}} {2} {\l__celtic_width_int-1}
{
  \tl_if_exist:cT {crossing ####1 - ##1}
  {
    \tl_if_eq:cNTF {crossing ####1 - ##1} \l__celtic_bar_tl
    {
```

Vertical crossing.

```
      \path[celtic~ bar/.try] (####1,##1-1) -- (####1,##1+1);
    }
    {
```

Horizontal crossing.

```
      \path[celtic~ bar/.try] (####1-1,##1) -- (####1+1,##1);
    }
  }
}
  }
```

Now we work through the crossings, trying to generate a path starting at each one. The crossings are at points (x, y) with $x + y$ odd.

```
  \int_step_inline:nnnn {1} {1} {\l__celtic_height_int-1}
  {
    \int_step_inline:nnnn {1 + \int_mod:nn {##1}{2}} {2} {\l__celtic_width_int-1}
{
```

Attempt to generate a path starting from that crossing. The third argument is to indicate which way the under-path goes from that crossing.

```
    \celtic_generate_path:nnx {####1}{##1}{\int_eval:n {\l__celtic_flip_int*(2*\int_mod:nn{###
    }
  }
```

Once we have generated our paths, we render them and close our scope and group.

```
  \celtic_render_path:
  \endpgfscope
  \group_end:
}
```

`\celtic_generate_path:nnn` This macro generates a sequence of path segments.

```
\cs_new_nopar:Npn \celtic_generate_path:nnn #1#2#3
{
```

First off, we test to see if the given coordinates are allowed as a starting point. If the crossing has a wall or it is already marked as "used" then it isn't.

```
  \bool_if:nF {
    \tl_if_exist_p:c {crossing #1 - #2}
    ||
    \tl_if_empty_p:c {crossing used #1 - #2}
  }
  {
```

Those tests failed, so we procede. First, we mark the crossing as used and set our initial data. Position, original position, and direction.

```
    \tl_clear:c {crossing used #1 - #2}
    \int_incr:N \l__celtic_component_int
    \int_set:Nn \l__celtic_x {#1}
    \int_set:Nn \l__celtic_y {#2}
    \int_set_eq:NN \l__celtic_ox \l__celtic_x
```

```
    \int_set_eq:NN \l__celtic_oy \l__celtic_y
    \int_set:Nn \l__celtic_dx {#3}
    \int_set:Nn \l__celtic_dy {1}
```

This holds our recursion index so that we can bail out if we look like we're entering a loop (which we shouldn't).

```
    \int_zero:N \l__celtic_int
```

We stop the loop if we get back where we started or we hit the maximum recursion limit.

```
    \bool_do_until:nn
    {
      (\int_compare_p:n {\l__celtic_x == \l__celtic_ox}
      &&
      \int_compare_p:n {\l__celtic_y == \l__celtic_oy}
      )
      || \int_compare_p:n {\l__celtic_int > \l__celtic_max_steps_int}
    }
    {
```

Increment our counter.

```
      \int_incr:N \l__celtic_int
```

Create the segment between this crossing and the next one.

```
      \celtic_next_crossing:
```

Store the segment, its over-crossing, and its component number. Then return to the start of the loop.

```
      \seq_put_left:NV \l__celtic_path_seq \l__celtic_path_tl
      \seq_put_left:NV \l__celtic_crossing_seq \l__celtic_crossing_tl
      \seq_put_left:NV \l__celtic_component_seq \l__celtic_component_int
    }
```

If we hit the maximum number of steps, issue a warning.

```
    \int_compare:nT {\l__celtic_int > \l__celtic_max_steps_int}
    {
      \msg_warning:nn {celtic} { max~ steps }
    }
  }
}
```

`\celtic_generate_path:nnx` Useful variant.

```
\cs_generate_variant:Nn \celtic_generate_path:nnn {nnx}
```

`\celtic_render_path:` This takes a generated list of path segments and renders them.

```
\cs_new_nopar:Npn \celtic_render_path:
{
```

First pass through the sequence of segments.

```
  \seq_map_inline:Nn \l__celtic_path_seq
  {
```

We need to get the component number, but pop removes it from the sequence so we put it back at the other end again.

```
    \seq_pop:NN \l__celtic_component_seq \l__celtic_tmpa_tl
    \seq_put_right:NV \l__celtic_component_seq \l__celtic_tmpa_tl
```

Draw the path segment, styling by the component number.

```
    \path[celtic~ path/.try, celtic~ path~ \tl_use:N \l__celtic_tmpa_tl/.try] ##1;
  }
```

This next bit of code attempts to work out the true thickness of the presumably doubled path. We do it in a group and scope to limit its effect.

```
  \group_begin:
  \pgfscope
  \tikzset{celtic~ path/.try}
  \tl_use:c {tikz@double@setup}
```

This gets the resulting line width outside the group and scope.

```
  \tl_set:Nn \l__celtic_tmpa_tl
  {
    \endpgfscope
    \group_end:
    \fp_set:Nn \l__celtic_clip_fp
  }
  \tl_put_right:Nx \l__celtic_tmpa_tl {{\dim_use:N \pgflinewidth}}
  \tl_use:N \l__celtic_tmpa_tl
```

Now we set the inner and outer clip sizes based on that line width.

```
  \fp_set:Nn \l__celtic_inner_clip_fp {sqrt(2) * (\l__celtic_clip_fp + \l__celtic_inner_fp)}
  \fp_set:Nn \l__celtic_clip_fp {sqrt(2) * (\l__celtic_clip_fp + \l__celtic_outer_fp)}
```

This second pass through the segments redraws each one clipped to a diamond neighbourhood of its over-crossing.

```
  \seq_map_inline:Nn \l__celtic_path_seq
  {
```

We get the crossing coordinate.

```
    \seq_pop:NN \l__celtic_crossing_seq \l__celtic_crossing_tl
```

Again, we need the component number.

```
    \seq_pop:NN \l__celtic_component_seq \l__celtic_tmpa_tl
    \seq_put_right:NV \l__celtic_component_seq \l__celtic_tmpa_tl
    \pgfscope
```

This is the smaller of the clip regions.

```
    \clip \l__celtic_crossing_tl +(-\fp_to_dim:N \l__celtic_inner_clip_fp,0) -- +(0,\fp_to_dim
```

We draw just the background part of the (presumably doubled) path.

```
    \path[celtic~ path/.try, celtic~ path~ \tl_use:N \l__celtic_tmpa_tl/.try, double~ backgrou
    \endpgfscope
    \pgfscope
```

Noew we apply the larger clip region.

```
    \clip \l__celtic_crossing_tl +(-\fp_to_dim:N \l__celtic_clip_fp,0) -- +(0,\fp_to_dim:N \l_
```

And draw the foreground part.

```
    \path[celtic~ path/.try, celtic~ path~ \tl_use:N \l__celtic_tmpa_tl/.try,double~ foreground
    \endpgfscope
  }
}
```

We are now leaving LaTeX3 world.

```
\ExplSyntaxOff
```

Clipping with doubled paths isn't perfect when anti-aliasing is used as it produces artefacts where the lower path shows through. To get round that, we need to draw the two parts of the doubled path separately. The following two keys extract the line widths and colours of the two parts of a doubled path and apply it.

```
\tikzset{
```

This sets the stye to that of the under path.

```
  double background/.code={%
    \begingroup
    \tikz@double@setup
    \global\pgf@xa=\pgflinewidth
    \endgroup
    \expandafter\tikz@semiaddlinewidth\expandafter{\the\pgf@xa}%
    \tikz@addmode{\tikz@mode@doublefalse}%
  },
```

This to the over path.

```
  double foreground/.code={%
    \begingroup
    \tikz@double@setup
    \global\pgf@xa=\pgfinnerlinewidth
    \endgroup
    \expandafter\tikz@semiaddlinewidth\expandafter{\the\pgf@xa}%
    \tikz@addmode{\tikz@mode@doublefalse}%
    \tikzset{color=\pgfinnerstrokecolor}%
  },
}
```