qrcodetikz: prettier QR codes

Miguel Vinícius Santini Frasson
mvsfrasson@gmail.com

2025–05–28, version 1.0

Contents

1 Introduction

The package `qrcodetikz` aims to improve the display of QR codes provided by package `qrcode`.

with `qrcode` vs. with `qrcodetikz`

The Quick Response (QR) codes provided by package `qrcode` show white borders on each square (from little to very proeminent depending on pdf viewer). This is because the QR code is printed square by square, not the connected regions of squares as such, and pdf screen viewers show these undesired square borders.

This package overwrites the `qrcode` printing functions to fill connected regions of the QR code using Ti*k*Z, allowing prettier qrcodes on screen visualization, with possibility of customization.

2 Usage

To switch the display of QR codes with Ti*k*Z, just load package `qrcodetikz` instead of `qrcode`. The package options are the same of `qrcode` package[1], that we repeat briefly (for details, see `qrcode` documentation):

nolinks disable qrcode links if `hyperref` is loaded.

draft doesn't compute QR codes, dummy QR codes are displayed, like

final opposite to draft, display complete QR codes.

forget do not store QR codes into `.aux` for reusage, recompute them in every run (mainly for debug purposes).

You can set other options using `\qruse{⟨options⟩}`, which can be set as options to `\qrcode` command. These options are:

height=⟨*dimension*⟩ sets the height (and width) of the QR codes. Default is 2 cm;.

level=⟨*L-M-Q-H*⟩ sets the level of error correction; the levels are set with one letter: `L` (low), `M` (medium), `Q` (quality) and `H` (high). Default is `M`.

version=⟨*1-to-40*⟩ is related to the size if the QR code. Must be a number between 1 and 40. Default is the lowest version that encodes text with desired quality.

padding, tight QR code specification says that a QR code should be surrounded by some white space (specifically the width of 4 modules). With `padding`, this white border is added. With `tight`, this white border is omitted, since it is usually provided by the user. Defult to `tight`.

link, nolink Allows/disallows use of clickable links with the QR code.

For many more details, like special characters, please refer to the documentation of `qrcode` package.

2.1 Fill options

`\qrcodeFillOptions` The default plain QR codes are fine and recommended. However, with `qrcodetikz`, the user can pass fill options `\fill` command to display the QR codes, like colors, rounded corners, shades, etc. using

`\qrcodeFillOptions{⟨fill-options⟩}`

The settings persist until the next call of `\qrcodeFillOptions` (for example, a call of `\qrcodeFillOptions{}` erases previous customizations). As an example:

```
\begin{center}
  % rounded corners
  \qrcodeFillOptions{rounded corners=1pt}
  \qrcode{QR codes}\qquad
  % shades
```

[1] All options are passed to `qrcode` package.

```
  \qrcodeFillOptions{left color=gray,right color=black,draw=black,thin}
  \qrcode{QR codes}\qquad
  % removing customization
  \qrcodeFillOptions{}
  \qrcode{QR codes}
\end{center}
```

would print:

Notice that last qrcode was printed plain, without customizations.

In our humble opinion, plain QR codes are prettier. The implementation made an effort to produce good looking QR codes with fill option `rounded corners`, filling individual connected components of the QR code to avoid straight corners. Other customizations with fill options are left for the convenience of the user.

`\qrcodetikzOn` `\qrcodetikzOff` One can turn on and off QRcodes with Ti*k*Z with commands `\qrcodetikzOn` and `\qrcodetikzOff`. When turned off, original `qrcode` functions for printing are restored. These functions probably will never be used outside this manual ☺.

This package was not intended specifically to display artistic QR codes. For that you can take a look into `fancyqr` package.

For the regular user, the documentation ends here. In the following pages we give details of the implementation.

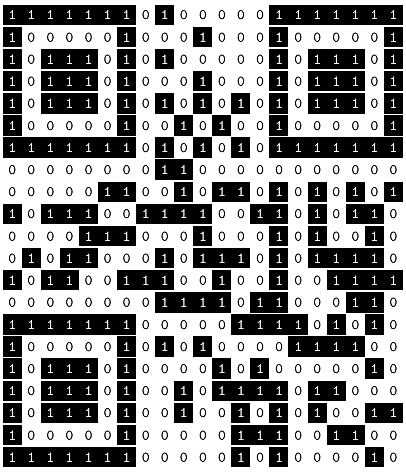
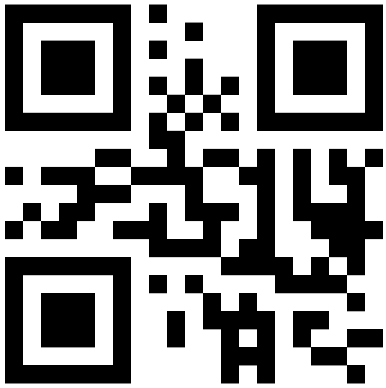
3 Implementation

3.1 Ideas and conventions

Using the `qrcode` package, that saves computed QR codes into the `.aux` file, if one calls `\qrcode{QrCode}`, we get into the `.aux` file the line below (as a long line without linebreaks):

```
\qr@savematrix{QrCode}{1}{2}{1111111010000011111111000001000100010000011011
1010100000101110110111010001000101110110111010101010101110110000010010100 10
0000111111110101010111111100000000110000000000000000011001011010101011011100
1111001101011000001110001000101001001011000101110101111010110011100100100 11
1100000000111101100011011111110000011110101010000010101000011110010111010 00
0101000001010111010010111101100010111010010010101001110000010000011100110 01
1111110000010100 0010}
```

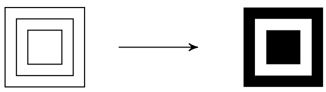
Counting the 0's and 1's, we get $441 = 21^2$ "bits". Dividing the string into 21 lines of 21 bits, coloring 1's as black boxes and 0's as light boxes to ease visualization we get:

(printed with `qrcode`)

So one sees that the binary data must be printed in rows from left to right, from top to bottom. On the right of the picture above it is the QR code as printed with the `qrcode` package: it prints individual black squares for each 1 and equivalent space for 0. An unpleasant side effect are the visible thin borders of squares when pdf file is visualized on screen. This package is meant to fix this, printing regions as a whole with Ti*k*Z.

We observed that if we make a path connecting all connected borders of the pixels, and fill this path with the "even-odd rule" [2], it would beautifully print the QRcode.



Fill with even-odd rule.

This package is the implementation of such idea. We took advantage of facilities from LaTeX3 such as loops, sequences (lists), arrays of integers, property lists, easy integer computations and other goodies already loaded into the LaTeX kernel in every run in any recent TeX distribution. The only external packages needed are `qrcode` (to compute the qrcode) and `tikz` (to fill a path in even-odd rule).

If any other package wants to take advance of the conversion (binary data) ↔ (Ti*k*Z path), for artistic reasons for instance, could use

`\QRTZgetTikzPathMaybeSaved{⟨binary data⟩}{⟨cmd to store path⟩}`

explained in section 3.8, that tries to reuse computed paths or computes it otherwise. The first argument is expanded, so it can be a command that holds the binary data.

In QRcode jargon, the size in number of squares of the QR code is given by its *version*. This size is $4 \times \langle \textit{qrcode version} \rangle + 17$. The maximum version os 40, so the maximum size is $4 \times 40 + 17 = 177$. We take advance of integer arrays in LaTeX3 (module l3intarray) that provide fast access to any integer in the array and store all matrices $n \times n$ as linear sequence of n^2 integers, in the order as the binary data.

[2] The SVG defines the even-odd rule by saying: "This rule determines the 'insideness' of a point on the canvas by drawing a ray from that point to infinity in any direction and counting the number of path segments from the given shape that the ray crosses. If this number is odd, the point is inside; if even, the point is outside." (`https://en.wikipedia.org/wiki/Even-odd_rule`).

As Ti*k*Z uses cartesian coordinates, we decided to use them as the coordinates of the pixels. We implement the conversion from (x, y) coordiante, with x and y from 1 to n, to sequence index from 1 to n^2 so that

- pixel $(1, n)$ – top left – is in index 1;
- pixel (n, n) – top right – is in index n;
- pixel (n, n) – bottom right – is in index n^2;
- pixel $(1, 1)$ – bottom left – is in index $(n - 1)n + 1$.

Doing the math, index is $(n - y)n + x$.

We need to put a convention to the coodinates of the corners. We define that a pixel (x, y) gives its coordinates to the bottom-left corner. The coordinates of the lines vary from 1 to $n + 1$. This way, the size of the smallest line segment is 1 and the size of a QR code is a square of size n, so the path must be scaled down to fit the required size of the QR code.

With these conventions, we can begin implementation according the following steps:

1. We store the binary matrix into a array of integers; we use intarrays from LaTeX3 to get fast read/write access. The interface $(x, y) \leftrightarrow i$ (array index) is explained in section 3.4. At this point, every pixel is stored as values 0 (for empty) or 1 (for black).

2. We compute the connected components of the QR code using "find/union" operations, a fast algorithm, explained in section 3.5. The content of the pixel will be 0 (for empty) or a positive integer that labels the connected component, shared by all pixels in that same connected component. A positive value will be printed as black.

3. We build a list of all borders of pixels. We have to fix notation for a border: after some reimplementations[3], we came up with the notation of border as a comma list x, y, d, c where (x, y) is the origin of the border, d is a direction number from 0 to 3 (representing angle $d \cdot 90°$): 0 ($\rightarrow$), 1 ($\uparrow$), 2 ($\leftarrow$), 3 ($\downarrow$) in such way that the black pixel is always to the left of the border and c is the number of the connected component that such pixel belongs to. Following a border to the next in the same connected component will walk the border in "positive" direction (counterclockwise), and in a crossing, turn left (that is, add 1 modulus 4) will always turn inward. The process of creating the list of borders is described in section 3.6.

4. Now we build the Ti*k*Z path, the main part of the package. Starting from any border, the next border is either turn left (inward), go straight or turn right in the same connected component, trying in this sequence. Every walked border is removed from the list of borders. In the beginning and at every turn, output "(x, y)--" to the path, where (x, y) is the origin of the border. When none of these borders is in the list, we arrive at a dead end and output

[3] After first successful implementation, we realized that `rounded corners` fill option produced ugly straight crossings so we decided to implement connected components and follow the path always inward in a crossing with multiple ways. A natural choice as to walk the borders in "positive" sense according to Calculus rules, that is, interior is always on the left, so inward means turn left in most cases. After some reasoning, we came up with the directions 0 to 3, so turn left means add 1 modulus 4 and turn right means add -1 modulus 4.

"cycle" to the path. This pocess is reeated while the list of borders is not empty. This algorithm is implemented in section 3.7.

5. The package `qrcode` saves the correspondence "text+version" ↔ "binary data" into the aux file, and at beginning of document, this code saves the binary data as a macro, avoiding most of the computation. For the computed Ti*k*Z paths we implement this saving/restoring as a LaTeX3 prop list, also saving the correspondence "binary data" ↔ "path" into the aux file. This is done in section 3.8.

6. In the mood of `qrcode` package, we save into the aux file a line that populates the prop list from previous item. This is done in section 3.9.

7. In section 3.10 we implement functions that are meant to replace `qrcode` functions: that package implements matrices as a series of macros. Fortunately `qrcode` has a function to convert its matrices as binary data. To avoid any change of behavior and get the closest result from `qrcode`, we copied most of the computations plainly from that package, but printing the path with Ti*k*Z instead of the original display of individual boxes.

8. Finally, in section 3.11 we save copies of former functions and implement turning on and off of new printing functions.

3.2 Package identification

This package uses LaTeX3 interface, with most recently implemented functions in l3intarray module from 2018. In 2020[4], LaTeX3 was included into LaTeX 2_ε, so we require LaTeX 2_ε version at least as recent as 2020. If you only have access to ealier versions of LaTeX 2_ε, you can use `\usepackage{expl3}`, but we still depend on LaTeX3 *intarray* module from 2018.

This package passes all options to `qrcode` package and loads it, if not already loaded.

```
\NeedsTeXFormat{LaTeX2e}[2020-02-02]
\ProvidesPackage{qrcodetikz}[2025-05-28 v1.0 Prettier qrcodes]
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{qrcode}}
\ProcessOptions
\RequirePackage{qrcode}
\RequirePackage{tikz}
```

The package uses LaTeX3 in most of its extension. Only the final printing functions that will replace `qrcode` printing functions don't use LaTeX3.

```
\ExplSyntaxOn
```

3.3 Declaration of variables

We declare some few variables. Variables of type "token list" should end with the suffix `_tl`, but we decided to ommit it for readability.

- Integer variable `\l_qrtz_size_int` holds the size of the side of QR code.

```
    \int_new:N \l_qrtz_size_int
```

[4]See `https://www.latex-project.org/news/latex2e-news/`, entry for 2020/02/02.

- Individual coordinates of (x, y) are stored in

```
\tl_new:N \l_qrtz_x
\tl_new:N \l_qrtz_y
```

- The direction number (0 to 3) is stored in

```
\tl_new:N \l_qrtz_dir
```

- The number of current connected component is stored in

```
\tl_new:N \l_qrtz_component
```

- Current path in construction is stored in

```
\tl_new:N \l_qrtz_path
```

- Borders are stored in

```
\tl_new:N \l_qrtz_border
\tl_new:N \l_qrtz_inner
\tl_new:N \l_qrtz_straight
\tl_new:N \l_qrtz_outer
```

- LaTeX3 intarray that stores matrix. Its size is $177^2 = 31\,329$, the worst possible case.

```
\intarray_new:Nn \g_qrtz_labels_intarray {177 * 177}
```

- LaTeX3 sequence to store the list of borders

```
\seq_new:N \l_qrtz_border_seq
```

- LaTeX3 property list that stores saved information (binary-data ↔ path correspondence)

```
\prop_new:N \g_qrtz_paths_prop
```

- Switch for fallback definition of .aux commands

```
\bool_new:N \g_qrtz_aux_fallback_written_bool
\bool_gset_false:N \g_qrtz_aux_fallback_written_bool
```

- Auxiliary switches

```
\bool_new:N \l_qrtz_continue_straight_bool
\bool_new:N \l_qrtz_can_continue_bool
```

3.4 Interface for matrix and intarray correspondence

`\qrtz_index:nn` `\qrtz_pixel:nn`

Pixel (x, y) corresponds to index $(n - y)n + x$ on the intarray. Function `\qrtz_index:nn {`$\langle x \rangle$`}{`$\langle y \rangle$`}` returns the linear index of a pixel in coordinates (x, y). Size is stored in `\l_qrtz_size_int`. Function `\qrtz_pixel:nn {`$\langle x \rangle$`}{`$\langle y \rangle$`}` returns the value of the entry, that is, 0 for white pixel, positive (the connected component number) if black.

```
\cs_new:Nn \qrtz_index:nn {
  ( \l_qrtz_size_int - (#2) ) * \l_qrtz_size_int + (#1)
}
%
\cs_new:Nn \qrtz_pixel:nn {
  \intarray_item:Nn \g_qrtz_labels_intarray { \qrtz_index:nn {#1}{#2} }
}
```

3.5 Computation of connected components

We use *union-find algorithm* to compute connected components. The idea is that every pixel is a node in a graph, and a pointed connection $A \to B$ means that "B is parent of A". If A points to itself, it means that A is a "patriarch". Two nodes A and B belong to the same family if they share the same patriarch.

For example, consider the pointed graph

$$\begin{array}{c} 3 \\ \downarrow \\ 1 \to \ 2 \ \to 4 \circlearrowleft, \quad 5 \circlearrowleft, \quad 6 \to 7 \to 8 \circlearrowleft \end{array}$$

The patriarchs are 4, 5 and 8. The families (*i.e.*, connected components) are $\{1, 2, 3, 4\}$, $\{5\}$ and $\{6, 7, 8\}$.

There are two functions: *find* that returns the patriarch of an element, and *union* that joins two families, making a former patriarch to point another node in another family (otherwise we get a cycle), joining the families under one of the patriarchs. A way to avoid cycles is to make a patriarch to point another patriarch.

We implement the "pointing" as the result of an array $A[i] = j$, meaning $i \to j$.

The graph above is represented by the array:

i	1	2	3	4	5	6	7	8
$A[i]$	2	4	2	4	5	7	8	8

- find(i): $j \leftarrow i$. while $A[j] \neq j$ do: $j \leftarrow A[j]$. Return j.
- union(i, j): $A[\text{find}(i)] \leftarrow \text{find}(j)$.

We initialize the array with $A[i] = i$ if pixel is black or $A[i] = 0$ otherwise, for all i (all black pixels are disconnected).

First pass: Loop through all pixels x, y, that correspond to index i. If a pixel p is black: if pixel q to the right of p is also black, perform union(p, q); if pixel q below p is also black, perform union(p, q).

At the end of first pass, all connected components are determined. It remains to clearly label the component of each pixel, doing the second pass:

Second pass: loop for all indexes i: $A[i] \leftarrow$ find(i).

Now each pixel p, which has index i[5], belongs to component $c = A[i]$. Below we implement that. The array A is the variable `\g_qrtz_labels_intarray`.

`\qrtz_find:nnN` Implementation of find: `\qrtz_find:nnN {`$\langle x\rangle$`}{`$\langle y\rangle$`} #3`. Last arg `#3` should be an integer var (module l3int) that will receive the index of the patriarc of (x, y).

```
\cs_new:Nn \qrtz_find:nnN {
  \int_set:Nn #3 { \qrtz_index:nn {#1}{#2} }
  \int_until_do:nNnn {\intarray_item:Nn \g_qrtz_labels_intarray #3} = #3
  {
    \int_set:Nn #3 {\intarray_item:Nn \g_qrtz_labels_intarray #3}
  }
}
```

`\qrtz_union:nnnn` Implementation of *union* of pixels (x_0, y_0) and (x_1, y_1).

```
\cs_new:Nn \qrtz_union:nnnn {
  \qrtz_find:nnN {#1}{#2} \l_tmpa_int
  \qrtz_find:nnN {#3}{#4} \l_tmpb_int
  \intarray_gset:Nnn \g_qrtz_labels_intarray \l_tmpb_int \l_tmpa_int
}
```

`\qrtz_compute_components:n` A function that performs the two passes of the algorithm described above. Its argument `#1` is the binary data of the QR code, loaded into the intarray `\l_qrtz_labels_intarray`: 1 becomes the index i, 0 remains 0. Index will be stored in `\l_tmpa_int`

```
\cs_new:Nn \qrtz_compute_components:n {
  \int_zero:N \l_tmpa_int
  \tl_map_inline:nn {#1}
  {
    \int_incr:N \l_tmpa_int
    \intarray_gset:Nnn
    \g_qrtz_labels_intarray { \l_tmpa_int } { ##1 * \l_tmpa_int }
  }
```

Set matrix size n as $n \leftarrow \sqrt{l}$, where $l = length$ is stored in `\l_tmpa_int` after previous loop.

```
  \int_set:Nn \l_qrtz_size_int { \fp_to_int:n { sqrt( \l_tmpa_int ) } }
```

First pass: Iterate x,y from 1 to `\l_qrtz_size_int` (size of side of QRcode); if (it make sense pixel at right of (x, y)) *and* (current pixel is black) *and* (pixel to the right is black) ["and" computed with `lazy_all`, that is, one false condition stops evaluation, returns false], do union operation of these pixels. The analogue conditions for pixel below current pixel.

```
  \int_step_inline:nn { \l_qrtz_size_int } { % i=##1
    \int_step_inline:nn { \l_qrtz_size_int } { % j = ####1

      \bool_if:nT {
        \bool_lazy_all_p:n {
          { \int_compare_p:nNn {##1} < \l_qrtz_size_int }
          { \int_compare_p:nNn { \qrtz_pixel:nn {##1}{####1} } > 0 }
          { \int_compare_p:nNn { \qrtz_pixel:nn {1 + ##1}{####1} } > 0 }
        }
      }
      { \qrtz_union:nnnn {##1}{####1} {1 + ##1}{####1} }

```

[5] The index of a pixel (x, y) is given by function `\qrtz_index:nn {x} {y}`.

```
      \bool_if:nT {
        \bool_lazy_all_p:n {
          { \int_compare_p:nNn {####1} < \l_qrtz_size_int }
          { \int_compare_p:nNn { \qrtz_pixel:nn {##1}{####1} } > 0 }
          { \int_compare_p:nNn { \qrtz_pixel:nn {##1}{1 + ####1} } > 0 }
        }
      }
      { \qrtz_union:nnnn {##1}{####1} {##1}{1 + ####1} }
    }
  }
```

Second pass: if pixel is black, set its value as its patriarch (find operation). `\l_tmpa_int` holds index of current pixel, `\l_tmpb_int` holds index of its patriarch.

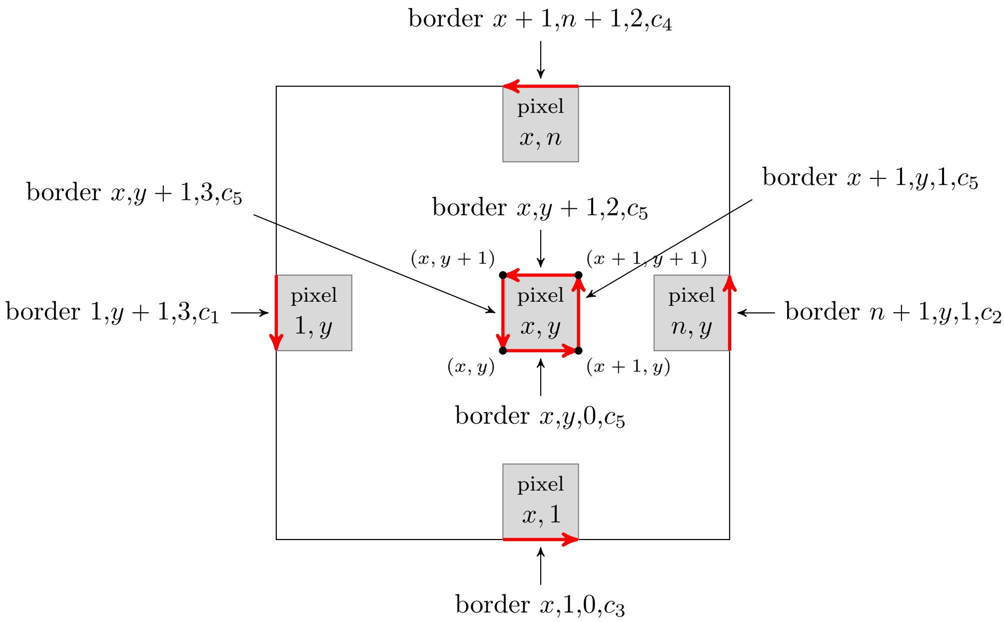
```
  \int_step_inline:nn { \l_qrtz_size_int } { %   i = ##1
    \int_step_inline:nn { \l_qrtz_size_int } { % j = ####1

      \int_set:Nn \l_tmpa_int { \qrtz_index:nn {##1} {####1} }

      \int_compare:nNnF
      {\intarray_item:Nn \g_qrtz_labels_intarray \l_tmpa_int} = 0
      {
        \qrtz_find:nnN {##1} {####1} \l_tmpb_int
        \intarray_gset:Nnn \g_qrtz_labels_intarray \l_tmpa_int \l_tmpb_int
      }
    }
  }
}
```

3.6 Construction of the list of borders

We decided that a border is a comma list x, y, d, c where (x, y) is the origin of the border, c is the number of the connected component of the pixel to which this border corresponds and d is one of the numbers 0, 1, 2 or 3 meaning the direction $d \cdot 90°$, with the convention that the interior (black pixel) in on the left of the border, so that when one walks by subsequent borders, the path is walked in positive sense, *i.e.*, counterclockwise.



`\qrtz_build_border_list:n`

We build the list of borders with the function `\qrtz_build_border_list:n`, whose arg `#1` is the binary data. It stores the borders in `\l_qrtz_border_seq` variable. Each border is a comma list ⟨*x*⟩,⟨*y*⟩,⟨*dir*⟩,⟨*component*⟩, as described above.

Loop for x, y from 1 to n. First `\int_step_inline:nn` for x, secont for y, so that x = `##1`, y = `####1`.

```
\cs_new:Nn \qrtz_build_border_list:n {
```

Next macro sets size and matrix from binary.

```
  \qrtz_compute_components:n { #1 }
  %
  \seq_clear:N \l_qrtz_border_seq
  %
  \int_step_inline:nn { \l_qrtz_size_int }
  {
    % x = ##1
    \int_step_inline:nn { \l_qrtz_size_int }
    {
      % y = ####1
```

Connected component number = `\l_qrtz_component`.

```
      \tl_set:Ne \l_qrtz_component { \qrtz_pixel:nn {##1}{####1} }
```

If pixel(x, y) is black (component > 0), check whether neighbor pixels are white. We use explicitly the feature of `\bool_lazy_or:nnT` that the second test is done only if first is false.

```
      \int_compare:nNnT { \l_qrtz_component } > 0
      {
```

For left borders: add border x,$y+1$,3,⟨*component*⟩

```
        \bool_lazy_or:nnT
```

```
        { \int_compare_p:nNn { ##1 } = 1 }
        { \int_compare_p:nNn { \qrtz_pixel:nn {##1 - 1}{####1} } = 0 }
        {
          \seq_put_left:Ne \l_qrtz_border_seq
          { ##1, \int_eval:n{ ####1 + 1 }, 3, \l_qrtz_component }
        }
        %
```

For right borders: add border $x + 1$,y,1,⟨*component*⟩

```
        \bool_lazy_or:nnT
        { \int_compare_p:nNn { ##1 } = { \l_qrtz_size_int } }
        { \int_compare_p:nNn { \qrtz_pixel:nn {##1 + 1}{####1} } = 0 }
        {
          \seq_put_left:Ne \l_qrtz_border_seq
          { \int_eval:n{ ##1 + 1 }, ####1, 1, \l_qrtz_component }
        }
        %
```

For top borders: add border $x + 1$,$y + 1$,2,⟨*component*⟩

```
        \bool_lazy_or:nnT
        { \int_compare_p:nNn { ####1 } = { \l_qrtz_size_int } }
        { \int_compare_p:nNn { \qrtz_pixel:nn {##1}{####1+1} } = 0 }
        {
          \seq_put_left:Ne \l_qrtz_border_seq
          {\int_eval:n{##1+1}, \int_eval:n{####1+1}, 2, \l_qrtz_component }
        }
        %
```

For bottom borders: add border x,y,0,⟨*component*⟩

```
        \bool_lazy_or:nnT
        { \int_compare_p:nNn { ####1 } = 1 }
        { \int_compare_p:nNn { \qrtz_pixel:nn {##1}{####1-1} } = 0 }
        {
          \seq_put_left:Ne \l_qrtz_border_seq
          { ##1, ####1, 0, \l_qrtz_component }
        }
      }
    }
  }
}
```

3.7 Computation of the path

Now we get to the main task of the package. Basicly, get a border, follow the path (always removing the "walked" borders) in the same connected component until get a closed path, output that subpath and repeat until list of borders is empty.

The algorithm is the following:

path ← " "

A while *border-list*
  can-cont ← true
  go-straight ← false   *% to force first output to path*
  get *border* in *border-list*   *% without removing it*
  set *x*,*y*,*dir*,*component* from *border*

```
[B]   while can-cont
        remove border     % got at the beginning or tested in the if's at end of loop
        if not go-straight    % at turn, output to path
        | path ← path + "(x, y)--"
        end-if
        set x,y as the end point from border     % walk the border
[C]     compute inner,outer,straight borders
        % inner = x,y,dir+1,c,    straight = x,y,dir,c,    outer = x,y,dir–1,c,
        if inner in border-list
        | go-straight ← false
        | border ← inner
        else
        | if straight in border-list
        | | go-straight ← true
        | | border ← straight
        | else
        | | if outer in border-list
        | | | go-straight ← false
        | | | border ← outer
        | | else
        | | | can-cont ← false   % dead end, restart process to new subpath
        | | | path ← path + "cycle"
        | | end-if
        | end-if
        end-if
      end-while
    end-while
```

`\qrtz_build_tikz_path:nN`

We implement the construction of the path in function `\qrtz_build_tikz_path:nN`. The args are `#1` = ⟨*binary data*⟩ and `#2` = command that will hold the path.

```
\cs_new:Nn \qrtz_build_tikz_path:nN {
  \message{<Computing~ Tikz~ path~ ...}
  % build border list
  \qrtz_build_border_list:n { #1 }
```

Now the main action.

Initializing `path`

```
  \tl_set:Nn \l_qrtz_path {}
```

First loop at point [A]

```
  \bool_until_do:nn { \seq_if_empty_p:N \l_qrtz_border_seq }
  {
```

First code after [A]: initialize *can-cont*, *go-straight*, get border and set *x*, *y*, *dir* and *component* from it. The value false to *go-straight* forces first output to *path* that must happen at beginning of new subpath.

```
    \bool_set_true:N \l_qrtz_can_continue_bool
    \bool_set_false:N \l_qrtz_continue_straight_bool
    \seq_get_left:NN \l_qrtz_border_seq \l_qrtz_border
    \tl_set:Ne \l_qrtz_x         { \clist_item:Nn \l_qrtz_border 1 }
    \tl_set:Ne \l_qrtz_y         { \clist_item:Nn \l_qrtz_border 2 }
```

```
    \tl_set:Ne \l_qrtz_dir       { \clist_item:Nn \l_qrtz_border 3 }
    \tl_set:Ne \l_qrtz_component { \clist_item:Nn \l_qrtz_border 4 }
```

Second loop at point B: looping through a full subpath, walking borders continuously until this is not possible in thhe same connected component.

```
    \bool_while_do:Nn \l_qrtz_can_continue_bool
    {
```

Processing the border: remove from list of borders, update path if there was a turn in direction, "walk" the border, that is, update (x, y) as the end of that border.

```
     \seq_remove_all:NV \l_qrtz_border_seq \l_qrtz_border

     \bool_if:NF \l_qrtz_continue_straight_bool
     { % else
      \tl_put_right:Ne \l_qrtz_path { (\l_qrtz_x,\l_qrtz_y) -- }
     }

     \tl_set:Ne \l_qrtz_dir { \clist_item:Nn \l_qrtz_border 3 }

     \int_case:nn { \l_qrtz_dir }
     {
      0 { \tl_set:Ne \l_qrtz_x { \int_eval:n { \l_qrtz_x + 1 } } }
      1 { \tl_set:Ne \l_qrtz_y { \int_eval:n { \l_qrtz_y + 1 } } }
      2 { \tl_set:Ne \l_qrtz_x { \int_eval:n { \l_qrtz_x - 1 } } }
      3 { \tl_set:Ne \l_qrtz_y { \int_eval:n { \l_qrtz_y - 1 } } }
     }
```

Point C: compute all derivated borders that could continue the subpath: inner border has direction $dir + 1 \mod 4$, straight border has same dir and outer has direction $dir - 1 \mod 4$.

```
     \tl_set:Ne \l_qrtz_inner { % inner=left: dir+1 mod 4
      \l_qrtz_x, \l_qrtz_y,
      \int_eval:n { \int_mod:nn { \l_qrtz_dir + 1 } { 4 } },
      \l_qrtz_component
     }

     \tl_set:Ne \l_qrtz_straight {
      \l_qrtz_x, \l_qrtz_y, \l_qrtz_dir, \l_qrtz_component
     }

     \tl_set:Ne \l_qrtz_outer { % outer=right: dir-1 +4 mod 4
      \l_qrtz_x, \l_qrtz_y,
      \int_eval:n { \int_mod:nn { \l_qrtz_dir + 3 } { 4 } },
      \l_qrtz_component
     }
```

Tests in block at point C: test if inner, straight or outer borders are in list of borders, setting *go-straight* accordingly and setting *border* as the found member. his border set now will be processed in next iteration of the loop at B.

```
     \seq_if_in:NVTF \l_qrtz_border_seq \l_qrtz_inner
     {
      \bool_set_false:N \l_qrtz_continue_straight_bool
      \tl_set_eq:NN \l_qrtz_border \l_qrtz_inner
     }
     { % else
```

```
        \seq_if_in:NVTF \l_qrtz_border_seq \l_qrtz_straight
        {
          \bool_set_true:N \l_qrtz_continue_straight_bool
          \tl_set_eq:NN \l_qrtz_border \l_qrtz_straight
        }
        { % else
          \seq_if_in:NVTF \l_qrtz_border_seq \l_qrtz_outer
          {
            \bool_set_false:N \l_qrtz_continue_straight_bool
            \tl_set_eq:NN \l_qrtz_border \l_qrtz_outer
          }
          { % else (dead-end)
            \bool_set_false:N \l_qrtz_can_continue_bool
            \tl_put_right:Nn \l_qrtz_path { cycle }
          } % fi
        } % fi
      } % fi
    } % end-while
  } % end-while
```

At this point `\l_qrtz_path` holds the path of QRcode. Set `#2` as it.

```
  \tl_set_eq:NN #2 \l_qrtz_path
  \message{~done>.^^J}%
}
```

Geneate variant `eN` to always expand binary data and make as non-expl alias of that.

```
\cs_generate_variant:Nn \qrtz_build_tikz_path:nN { eN }

\tl_set_eq:NN \QRTZBinaryToTikzPath \qrtz_build_tikz_path:eN
```

3.8 Save paths in a LaTeX3 prop list

`\QRTZgetTikzPathMaybeSaved`

The macro `\QRTZgetTikzPathMaybeSaved` is the main function to retrieve the path corresponding to a binary data. If it was already computed in a previous run (either just compuuted or stored in the aux file) this path is stored into a LaTeX3 property list (l3prop module) and retrieved by this function. It is a variant of the function below.

```
\cs_new:Nn \qrtz_get_tikz_path_maybe_saved:nN {
  \prop_get:NeN \g_qrtz_paths_prop { #1 } { #2 }
```

`qrcode` oftion `forget` is respected.

```
  \use:c {ifqr@forget@mode} \tl_set_eq:NN #2 \q_no_value \fi
  \quark_if_no_value:NTF #2
  {
    \qrtz_build_tikz_path:nN { #1 }{ #2 }
    \qrtz_save_write_path_to_aux:ee { #1 }{ #2 }
  }
  {
    \message{<Using~ saved~ Tikz~ path>^^J}%
  }
}

\cs_generate_variant:Nn \qrtz_get_tikz_path_maybe_saved:nN { eN }
```

```
\tl_set_eq:NN \QRTZgetTikzPathMaybeSaved \qrtz_get_tikz_path_maybe_saved:eN
```

3.9 Saving paths to aux file

\QRTZsavePath

The macro `\QRTZsavePath {⟨binary⟩} {⟨tikz path⟩}` associates key ⟨*binary*⟩ to value ⟨*tikz path*⟩, for later use, to avoid recomputation of path and write correspondence to aux file. It writes a fallback dummy definition of itself, in case this package is removed. Its definition is actually a variant of the macro below.

```
\cs_new:Nn \qrtz_save_write_path_to_aux:nn {
  \bool_if:NF \g_qrtz_aux_fallback_written_bool
  {
    \iow_shipout:cn { @auxout } { \providecommand{\QRTZsavePath}[2]{} }
    \bool_gset_true:N \g_qrtz_aux_fallback_written_bool
  }
  \message{<Writing~ Tikz~ path~ to~ aux~ file>^^J}%
  \iow_shipout:cn { @auxout } { \QRTZsavePath {#1}{#2} }
```

Verifying if binary data was already saved; save if necessary.

```
\prop_get:NnN \g_qrtz_paths_prop { #1 } \l_tmpa_tl
  \quark_if_no_value:NT \l_tmpa_tl
  {
    \message{<Saving~ Tikz~ path~ to~ memory~ for~ later~ use>^^J}%
    \prop_gput:Nnn \g_qrtz_paths_prop { #1 } { #2 }
  }
}

\cs_generate_variant:Nn \qrtz_save_write_path_to_aux:nn { ee }

\tl_set_eq:NN \QRTZsavePath \qrtz_save_write_path_to_aux:nn
```

3.10 Printing function for `qrcode` matrix and binary data

\QRTZprintQRmatrix

`\QRTZprintQRmatrix` is `qrcodetikz`'s counterpart of `\qr@printmatrix` from `qrcode`, that prints the QR code stored in a `qrcode` matrix. In that package, matrices are stored with macros formed by the matrix name and matrix indexes. Fortunately it has a function to convert these matrices to binary data, that we print with `\QRTZprintBinaryString`.

```
\cs_new:Nn \qrtz_print_qr_matrix:n {
  \message{^^J~(}
  \use:c{qr@matrixtobinary}{#1} % stores into \qr@binarymatrix@result
  \QRTZprintBinaryString{ \use:c{qr@binarymatrix@result} }
  \message{~)~}
}

\tl_set_eq:NN \QRTZprintQRmatrix \qrtz_print_qr_matrix:n
```

We will copy lots of chunks of `\qr@printsavedbinarymatrix` from `qrcode` package, so we decided to go in $\LaTeX\,2_\varepsilon$ mode, but missing LaTeX3, specially by the trouble of the chain of `\expandafter` below.

```
\ExplSyntaxOff
```

\qrcodeFillOptions
\QRTZ@extraFillOpts

Macro that stores fill options and a macro that sets it.

```
\newcommand{\QRTZ@extraFillOpts}{}
\newcommand{\qrcodeFillOptions}[1]{\gdef\QRTZ@extraFillOpts{#1}}
```

\QRTZprintBinaryString

Function `\QRTZprintBinaryString` is `qrcodetikz` counterpart of function `\qr@printsavedbinarymatrix`, the macro of `qrcode` that prints the QR code represented by a binary string. We copy from it the computations and the use of `\parbox`, to ensure the same behavior of `qrcode` printing functions. It supposes context of a call of `\qrcode` command: `\qr@size` holds size, `\qr@desiredheight`, `\qr@modulesize`, `\qr@minipagewidth` are TEX lengths and `\ifqr@tight` holds tight option.

```
\newcommand{\QRTZprintBinaryString}[1]{%
  \setlength{\qr@modulesize}{\qr@desiredheight}%
  \divide\qr@modulesize by \qr@size\relax
  \setlength{\qr@minipagewidth}{\qr@modulesize}%
  \multiply\qr@minipagewidth by \qr@size\relax
  \ifqr@tight
  \else
    \advance\qr@minipagewidth by 8\qr@modulesize
  \fi
```

After computations, we prepair the Ti*k*Z settings: get the path for the binary data in `#1` . . .

```
  \QRTZgetTikzPathMaybeSaved{#1}{\QRTZtikzPath}%
```

and compute the scale. `\QRTZtikzPath` holds a path that displays data on a square of size n = `\qr@size` cm that has to fit the length `\qr@desiredheight`. 1 in = 2.54 cm = 72.27 pt, so 1pt = 0.03514598 cm. Doing the math, we have the scale below.

```
  \pgfmathsetmacro\QRTZtikzScale{0.03514598*\qr@desiredheight/\qr@size}%
```

We embed fill options in `\QRTZ@extraFillOpts` expanded once together with scale and even odd rule options.

```
  \expandafter\def\expandafter\QRTZinternalFillOptions\expandafter{\expandafter
    [\QRTZ@extraFillOpts,scale=\QRTZtikzScale,even odd rule]}%
```

Using the same `\parbox` as `qrcode` we fill the path. With option `padding`, we enlarge the bounding box by 4 units: as the path contains data in the rectangle from $(1, 1)$ to $(n + 1, n + 1)$, we add coordinates $(-3, -3)$ and $(n + 5, n + 5)$ to path, enlarging the bounding box inside Ti*k*Z.

```
  \parbox{\qr@minipagewidth}{%
    \tikz
    \expandafter\fill\QRTZinternalFillOptions
    \ifqr@tight\else(-3,-3)(\qr@size+5,\qr@size+5)\fi
    \QRTZtikzPath ;%
  }%
}
```

3.11 Replacing qrcode printing functions by new functions

To implement switches to turn on and off the replacements, we save copies of original functions and define macros that set original functions accordingly.

```
\let\qr@printmatrixORIGINAL\qr@printmatrix
```

```
\let\qr@printsavedbinarymatrixORIGINAL\qr@printsavedbinarymatrix

\newcommand{\qrcodetikzOn}{%
  \let\qr@printmatrix\QRTZprintQRmatrix
  \let\qr@printsavedbinarymatrix\QRTZprintBinaryString
}

\newcommand{\qrcodetikzOff}{%
  \let\qr@printmatrix\qr@printmatrixORIGINAL
  \let\qr@printsavedbinarymatrix\qr@printsavedbinarymatrixORIGINAL
}

\qrcodetikzOn
```

We silently fix a typo in `qrcode` (there, `\def\qr@fivezeros{11111}`), a not-so-serious bug, used only to compute a penalty).

```
\def\qr@fivezeros{00000}%
```