Pixel♥Art

v1.0.2 2023-02-20

A LaTeX package to draw pixel-art pictures

Louis Paternault

https://framagit.org/spalax/pixelart

spalax(at)gresille(dot)org

This package defines macros to draw pixel-art pictures using LuaLaTeX.

Table of Contents

Part I.
Introduction

Installation instruction are given in section 3. Documentation about how to use this package (and examples) is given in part II. Most of the examples has been gathered in appendix A.

1. Note to non-LuaLaTeX users

Unfortunately, package pixelart does not works with LaTeX or XeLaTeX. If you cannot or don't want to use LuaLaTeX, you can:

- use the previous, unmaintained version of pixelart, named pixelart0, and included with this package;
- or, better, use pxpic.

2. License

This work may be distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3 of this license or (at your option) any later version.

3. Installation

3.1. Gnu/Linux Distribution

If applicable, the easiest way to get pixelart working is by installing it by your distribution package. In Debian (and Ubuntu, and surely other distributions that inherit from Debian) it is packaged in `texlive-pictures` since version `2017.20180103-1`. So you can install it by running:

```
sudo apt install texlive-pictures
```

3.2. LaTeX distribution

This package is included both in TeXLive and MiKTeX. It can be installed using their respective package managers.

3.3. Manual installation

- Download the lastest archive :

  Stable version `https://mirrors.ctan.org/graphics/pgf/contrib/pixelart.zip`

  Development version `https://framagit.org/spalax/pixelart/repository/archive.zip?ref=main`

- Unzip the archive.

- Move files `pixelart.sty` and `pixelart.lua` into a LaTeX path.

Part II.

Usage

4. Package options

The package accepts a single option.

`draft`

Loading the package with this options is equivalent to using option `draft` on every single call of command `\pixelart` (or `\tikzpixelart`). If your document contains a lot of those commands, using this option can speed up compilation. For instance, on my outdated computer, compilation of this document takes about 11 seconds with this option, and more than 16 seconds without.

See option `draft` (page 4) for more information.

5. Drawing pictures

`\pixelart[`⟨*options*⟩`]{`⟨*pixels*⟩`}`

This is the basic command to draw pixel-art. Its mandatory argument is a sequence of pixels, as characters which will be turned into colored squares (using a `tikzpicture`, from Till Tantau's TikZ). Its basic usage is given in figure 1 (page 4).

In argument `{`⟨*pixels*⟩`}`, dots `.` are converted to transparent pixels, spaces and line breaks delimit pixel rows, and any character defines a color, depending on option `colors`.

More information is given in section 5.1.

`pixels`

The art, as a list of pixels. See section 5.1.

`colors = {`⟨*options*⟩`}`

Pixel colors. See section 5.2.

```
I \pixelart[
    tikz={
      red,
      scale=.05,
      baseline=.3em,
    }
  ]{
    ..11..11..
    .11111111.
    1111111111
    1111111111
    1111111111
    .11111111.
    ..111111..
    ...1111...
    ....11....
  } \LaTeX
```

I LaTeX

Figure 1: Basic example

`draft` = `true|false`

Disable rendering of `\pixelart` and `\tikzpixelart` commands: on documents containing a lot of thoses commands, this can speed up compilation. The pixel art is replaced by a dummy drawing, which takes the same space as the pixel art it replaces.

This option can be applied to every single `\pixelart` or `\tikzpixelart` of the document by using `\usepackage[draft]{pixelart}`.

See an example in A.5.

`margin` = ⟨*length*⟩

Pixel-art pictures are *enclosed* inside a tikz `\clip` command. Most of the time, you won't notice it, but when playing with options `style` or `tikz`, your picture might extend outside usual borders, and part of it might be clipped out. Use the `margin` option to extend the clip area. See example A.13.

`squares`

If set, each pixel is drawn using a separate `\fill` command (more in section 5.3). This option is incompatible with `stack` (see section 5.5).

`stack`

If set, pixelart will try to draw several adjacent pixels of the same color using a single `\fill` command (more in section 5.4). This option is incompatible with `squares` (see section 5.5).

`style` = ⟨*keyword*⟩ Default: `pixelart`

Pixels are drawn using `\fill` commands in a `tikzpicture` environment. Those `\fill` commands are applied the `pixelart` style by default (as in: `\fill[pixelart] (0, 0) rectangle (1, 1);`

You can set another style by using this command. See an example in A.6.

`tikz = {⟨options⟩}`

A pixel art is a bunch of `\fill` commands in a `tikzpicture`. Arguments to this option are passed as-is as optional arguments to the `tikzpicture` environment. See examples in A.7.

You will probably use this option to make the pixel art smaller (it is, by default, probably bigger than what you want), and to change the baseline. See figure 1 for an example.

Note that a `\clip` command, which does not accept arguments, is used to delimit the picture. So you might get an error if your `tikz` option applies to `\clip` commands (for instance, by using `every path/.style=bla bla bla`, which would apply to the `\clip` command, and would raise an error). In such cases, you might want to use option `style` instead (see page 5).

`\tikzpixelart[⟨options⟩]{⟨coord⟩}{⟨pixels⟩}`

Use this command if you want to draw pixel art pictures *inside* a `tikzpicture` environment.

The `[⟨options⟩]` and `{⟨pixels⟩}` arguments have the same meaning as the corresponding arguments to the `\pixelart` command (excepted for `tikz`, which is passed as is as the argument of a `scope` environment). The `{⟨coord⟩}` is the coordinate of the bottom left corner of the pixel art. It is passed as is in a TikZ `scope` environment (`\begin{scope}[shift={<coord>}]`), so you can use anything that will be correctly parsed by TikZ.

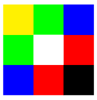
See examples in section A.10.

5.1. Pixels

Pixels are defined as blocks of characters. Dots always represent transparent pixels, and spaces and line breaks always separate pixel rows. Then, other caracters encode different colors, depending on option `colors`.

Let us analyse the following example.

```
\pixelart[
  colors=RGB, tikz={scale=.4}
]{
  {yellow}GB
  G.R
  BR{}
}
```



- The first row `{yellow}GB` means: one yellow pixel, one green pixel, one blue pixel.

- The second row `G.R` means: one green pixel, one transparent pixel, one red pixel.
- The last row `BR{}` means: one blue pixel, one red pixel, one pixel using the default TikZ color (that is, a `\fill` command is used to draw this pixel, without any `color` option, which means the default TikZ color is used; see option `tikz`, page 5).

Note that:

- Line breaks and consecutive spaces are considered as a *single* row separation. That is, the previous example could also have been written as `{yellow}B G.R      BR{}`.
- New paragraph are not allowed inside a `{`⟨*pixels*⟩`}` argument.
- If rows do not have the same lenght (the same number of pixels), transparent pixels are added at the end of the small rows.

Examples for the previous corner cases are given in section A.1.

Most of commands are expanded *before* being passed as an argument to `\pixelart`, so it is possible to use commands inside a `\pixelart{`⟨*pixels*⟩`}` argument. See section A.2 for an example.

5.2. Colors

In a `\pixelart` command, pixel colors can be defined explicitely, or using keys/values.

Explicit Any text surrounded by braces is transmitted as-is to the underlying `tikzpicture`. That way, you can use anything you would use with TikZ (which itself uses xcolor): `red`, `red!20`, `red!20!purple`, any user-defined color, or `{}`, which does not give any `color` argument to the `\fill` command used to draw this pixel, thus using default TikZ color (see option `tikz`, page 5).

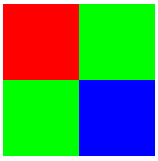
Key/Value Any alphanumeric character (`a` to `z`, `A` to `Z`, `0` to `9`) not surrounded by braces is a key, which encodes the corresponding color defined in the `colors` argument. For instance, if `colors` is given as `colors={R={red},G={green}}`, then, having `RG` in the `{`⟨*pixels*⟩`}` argument is equivalent to `{red}{green}`.

Using non-alphanumeric characters as keys of `colors` is not supported: it might work, but it is an undefined behaviour.

The following examples illustrates this option.

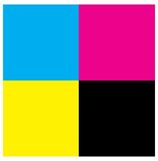
- Key/values:

```
\pixelart[
  colors={R=red, G=green, B=blue}
]{
  RG
  GB
}

```



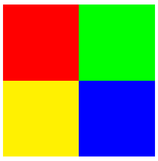
- Explicit:

```
\pixelart{
  {cyan}{magenta}
  {yellow}{black}
}

```



- Mix:

```
\pixelart[
  colors={R=red, G=green, B=blue}
]{
  RG
  {yellow}B
}

```



If you use the same setting several times, you can define it once for all using `\newpixelartcolors`.

`\newpixelartcolors{⟨name⟩}{⟨key/value⟩}`

Once a color set has been defined, to use it, simply use its name as an argument to `colors` instead of the whole keys/values. See an example in A.3.

`name`

Name of your color set.

`key/value`

Your color set, as keys/values.

`\renewpixelartcolors{⟨name⟩}{⟨key/value⟩}`

The previous command will fail if a color set `{⟨name⟩}` is already defined. This command will always succeed, whether a color set `{⟨name⟩}` already exists or not.

The following key/value settings are predefined (examples are given in A.4).

explicit : No keys/values: pixel colors have to be explicitely defined.

```
\newpixelartcolors{explicit}{}

```

RGB : Define the primary colors red, green, blue, as well as black and white.

```
\newpixelartcolors{RGB}{
  R=red,
  G=green,
  B=blue,
  W=white,
  K=black
}

```

BW : Black and white.

```
\newpixelartcolors{BW}{
  0=white,
  1=black,
}

```

gray : Gray scale: `0` is white, `9` is black, and digits `0` to `9` are a gray scale from white to black.

```
\newpixelartcolors{gray}{
  0=white,
  1=white!89!black,
  2=white!78!black,
  3=white!67!black,
  4=white!56!black,
  5=white!44!black,
  6=white!33!black,
```

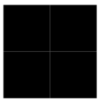


FIGURE 2: Artifacts appearing with option `squares` : the white lines should not exist.

```
      7=white!22!black,
      8=white!11!black,
      9=black,
    }

```

mono : Monochromatic: any character encodes the default TikZ color (black by default, may be changed with options `tikz` or `style`).

```
      \newpixelartcolors{mono}{
        a={},
        b={},
        c={},
        % ...
        9={},
      }

```

5.3. More about option `squares`

When using `squares`, each pixel is a separate colored square. Thus, artifacts can appear when viewing the document, as illustrated in figure 2.

To remove those artifacts, you can (1) use option `stack` instead (see section 5.4), or (2) use option `overlap`.

`overlap` = ⟨*number*⟩

Assuming the square width is 1, `{`⟨*number*⟩`}` is the size of the overlap: how much a square overlaps over the next one. With a positive number, the artifact (white line between two pixels) will disappear as the adjacent pixels are merged.

Note that using a negative value to `overlap` is also allowed, and will separate each pixel from its neighbour, giving a the look of a LCD screen.

See examples in section A.9.

		Drawing	
	mono	concentric	checker
pixelart0	243		
pixelart `stack`	2.9	4.4	158
pixelart `squares`	245	263	252

(a) File size (kB)

		Drawing	
	mono	concentric	checker
pixelart0	38.7		
pixelart `stack`	1.0	1.0	24.4
pixelart `squares`	19.9	28.3	27.1

(b) Compilation time (seconds)

FIGURE 3: Benchmark : pixelart0 vs. `stack` vs `squares`

5.4. More about option `stack`

If option `stack` is set, pixelart tries to merge adjacent pixels of the same color, to draw them at once. Thus, it mostly remove the artifacts problem discussed with option `squares` (although it may still occur).

Example in section A.8 shows how different color shapes are stacked onto each other to produce the pixel art.

5.5. Which option: `squares` or `stack`?

Which option should you use: `squares` or `stack`?

To compare the different algorithms, three files have been compiled, each with one big pixel art command (see example A.11), with three different algorithms: outdated package pixelart0, and options `stack` and `squares` of pixelart. The result can be seen in figure 3 (page 10).

Unless you want to use option `overlap` of option `squares` (see section 5.3), you probably want to use option `stack`:

- it is faster (see benchmark);
- it produces smaller files (see benchmark);
- it mostly remove artifacts.

Thus, the `stack` option is the default one used by pixelart.

6. Default options

You can use the following command to define options once for all.

`\setpixelartdefault{⟨options⟩}`
Any options defined here will affect later `\pixelart` or `\tikzpixelart` commands, unless those options are explicitely set.

See example in section A.12.

7. Logos

If you want to credit pixelart in a fancy way, you can use one of those logos. Apart from being gorgeous, they include a transparent text (so that they are searchable and copyable) and they scale with the current font size.

`\pixelartlogo`
Pixel♥Art Name and heart, in color.

`\pixelartheart`
♥ Heart only, in color.

`\pixelartname`
PixelArt Name only, in color.

`\pixelartlogobw`
Pixel♥Art Name and heart, black and white.

`\pixelartheartbw`
♥ Heart only, black and white

`\pixelartnamebw`
PixelArt Name only, black and white.

Note that those logos are affected by `\setpixelartdefault`, so, if you have used this command in your document, you might want to call `\setpixelartdefault{}` to reset the default options before drawing any logo.

8. Debugging

What if drawing your pixel art throws an error?

1. Standard output prints one `% pixelart 2, file ./foo.tex, input line 19` per `\pixelart` or `\tikzpixelart` command. If compilation failed right after this line, look at the given file: the `\pixelart` command might be wrong.

2. If you do not see the problem, set debug mode on.

   `\setpixelartdebugon`
   Set debug on.

   `\setpixelartdebugoff`
   Set debug off.

   Using those commands, the following piece of information will be logged:

   - the parsed options of `\pixelart` and `\tikzpixelart`;
   - the `{⟨pixels⟩}` argument of those commands;
   - the tikz commands used to draw the picture;
   - and maybe some other stuff I forgot.

Note that you can have as many `\setpixelartdebugon` and `\setpixelartdebugoff` commands as you want, even if debug is already on or off.

A good idea is to copy the tikz commands that are logged (those are the commands used to draw the pixel art), and paste them in a new file. Try to compile it. Are some of your arguments at fault? Bad value for the `tikz` option can produce faulty tikz code.

3. Still no idea why your `\pixelart` command fails? Seek help on your usual places, or report a bug.

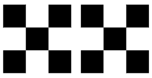
Part III.
Appendix

A. Examples

A.1. Corner cases of `\pixelart{`$\langle$*pixels*$\rangle$`}` argument

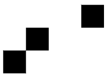
- Line breaks and multiple spaces are considered as a *single* row separation.

```
\pixelart{
  1.1
  .1.
  1.1
}
\pixelart{
  1.1                 .1.
  1.1
}
```



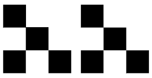
- Multiple line breaks (new paragraphs) are not supported: at best, they do not give the expected result; at worst, compilation will fail.

```
\pixelart{
  .1
  1.
}
\pixelart{
  .1

  1.
}
```



- Empty pixels are automatically added at the end of shorter lines.

```
  \pixelart{
    1
    .1
    1.1
  }
  \pixelart{
    1..
    .1.
    1.1
  }
```

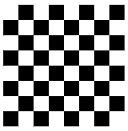


A.2. Commands inside `\pixelart{⟨pixels⟩}` argument

If we are too lazy to repeat every single pixel to draw a checker board, we can use commands.

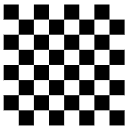
Example 1: Each couple of lines is repetead four times.

```
    \newcommand{\twolines}{
      .1.1.1.1 1.1.1.1.%
    }
    \pixelart{
      \twolines
      \twolines
      \twolines
      \twolines
    }

```



Example 2: `\foreach` outside `\pixelart`

```
\gdef\board{}
\gdef\line{1}
\foreach \j in {1, ..., 3}{
  \xdef\line{\line.1}
}
\foreach \i in {1, ..., 4}{
  \xdef\board{%
    \board
    \line.
    .\line.
  }
}
\pixelart{\board}
```



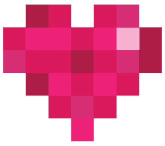
Example 3: I do not understand LaTeX enough to explain why, however, `\foreach` inside a `\pixelart` does not work.

```
\pixelart{
  \foreach \i in {1, ..., 8}{
    1\foreach \j in {1, ..., 3}{.1}.
    \foreach \j in {1, ..., 3}{.1}.1
  }
}
```

A.3. Define new color sets: `\newpixelartcolors`

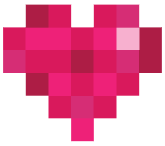
- Colors may be defined in the `colors` option

```
\definecolor{purple1}{RGB}{247, 176, 207}
\definecolor{purple2}{RGB}{238, 33, 120}
\definecolor{purple3}{RGB}{214, 45, 117}
\definecolor{purple4}{RGB}{217, 25, 92}
\definecolor{purple5}{RGB}{173, 29, 69}
\pixelart[colors={
    1=purple1, 2=purple2, 3=purple3, 4=purple4, 5=purple5,
}]{
  .54.43.
  4224215
  3445435
  .52424.
  ..434..
  ...2...
}
```



- Colors may be defined using `\newpixelartcolors`

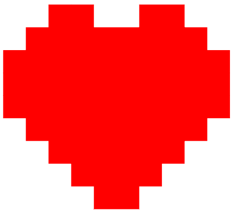
```
\definecolor{purple1}{RGB}{247, 176, 207}
\definecolor{purple2}{RGB}{238, 33, 120}
\definecolor{purple3}{RGB}{214, 45, 117}
\definecolor{purple4}{RGB}{217, 25, 92}
\definecolor{purple5}{RGB}{173, 29, 69}
\newpixelartcolors{purple}{
  1=purple1, 2=purple2, 3=purple3, 4=purple4, 5=purple5,
}
\pixelart[colors=purple]{
  .54.43.
  4224215
  3445435
  .52424.
  ..434..
  ...2...
}
```



A.4. Predefined color sets

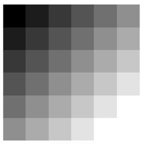
mono

```
  \pixelart[colors=mono, tikz={red, scale=.3}]{
      ..01..23..
      .456789ab.
      cdefghijkl
      mnopqrstuv
      wxyzABCDEF
      .GHIJKLMN.
      ..OPQRST..
      ...UVWX...
      ....YZ....
  }

```



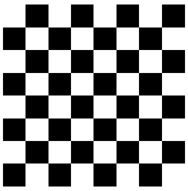
gray

```
  \pixelart[colors=gray, tikz={scale=.3}]{
      987654
      876543
      765432
      654321
      543210
      43210.
  }

```



BW

```
  \pixelart[colors=BW, tikz={scale=.3}]{
      01010101
      10101010
      01010101
      10101010
      01010101
      10101010
      01010101
      10101010
  }

```

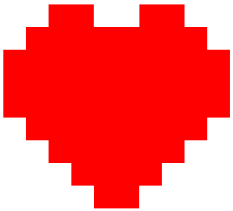


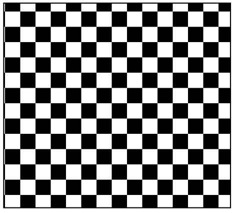
RGB

```
  \pixelart[colors=RGB, tikz={scale=.1}]{
    .....................KKKKKKK
    .RRR...GG..BBB..K..K.KWKKKWK
    .R..R.G..G.B..B.K.K..KWKKKWK
    .RRR..G....BBB..KK...KWKKKWK
    .R.R..G.GG.B..B.K.K..KWKWKWK
    .R..R.G..G.B..B.K..K.KWKWKWK
    .R..R..GG..BBB..K..K.KKWKWKK
    .....................KKKKKKK
  }

```

A.5. Option `draft`

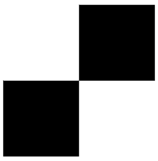
```
\setpixelartdefault{tikz={scale=.3}}

\pixelart[colors=RGB]{
  ..RR..RR..
  .RRRRRRRR.
  RRRRRRRRRR
  RRRRRRRRRR
  RRRRRRRRRR
  .RRRRRRRR.
  ..RRRRRR..
  ...RRRR...
  ....RR....
}

\pixelart[colors=RGB, draft]{
  ..RR..RR..
  .RRRRRRRR.
  RRRRRRRRRR
  RRRRRRRRRR
  RRRRRRRRRR
  .RRRRRRRR.
  ..RRRRRR..
  ...RRRR...
  ....RR....
}
```



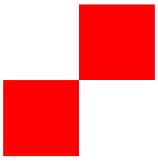


A.6. Option `style`

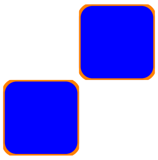
```
\pixelart{
  .1
  1.
}
```



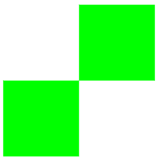
```
\pixelart[tikz={pixelart/.style={red}}]{
  .1
  1.
}
```



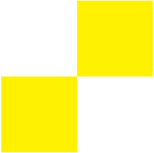
```
\tikzset{pixelart/.style={blue, draw=orange, very thick, rounded corners=5}}
\pixelart{
  .1
  1.
}
```



```
\pixelart[tikz={pixelart/.style={green}}]{
  .1
  1.
}
```

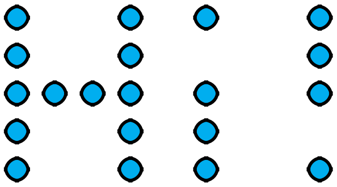


```
\tikzset{myfancystyle/.style={yellow}}
\pixelart[style=myfancystyle]{
  .1
  1.
}
```



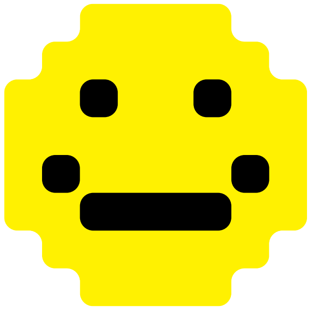
A more complex example.

```
\tikzset{pixelart/.style={cyan,
  very thick, draw=black, rounded
  corners=5}}
\pixelart[squares={overlap=-.2},
  colors=mono, tikz={scale=.5},
  margin=.1]{
  1..1.1..1
  1..1....1
  1111.1..1
  1..1.1...
  1..1.1..1
}
```



A.7. Option `tikz`

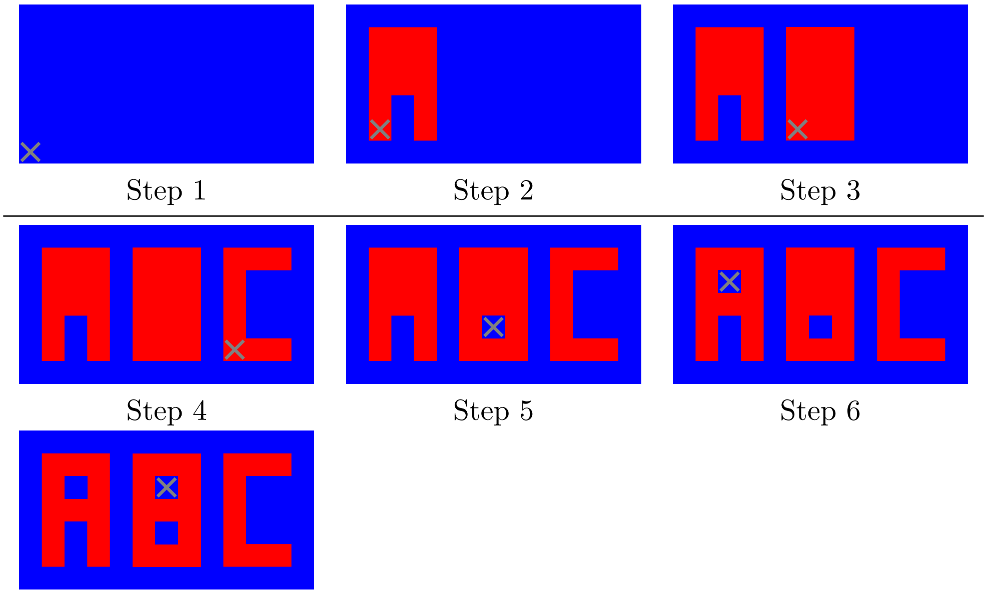
```
\pixelart[stack, colors={Y=yellow,
  K=black}, tikz={scale=.5, rounded
   corners=5}]{
  ..YYYY..
  .YYYYYY.
  YYKYYKYY
  YYYYYYYY
  YKYYYYKY
  YYKKKKYY
  .YYYYYY.
  ..YYYY..
}
```



A.8. Drawing steps with the `stack` option

Here is how an image is built with the `stack` option: each of the following steps is a `\fill` command with carefully crafted shape. Those filled area are "stacked" on each other, to produce the expected result.

Line per line, the algorithm checks if the pixels have been drawn yet. In the following steps, the gray cross marks the first pixel that was not drawn yet, that triggered a new `\fill` command.



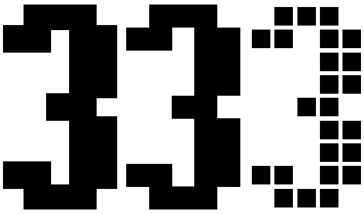
Step 1 Step 2 Step 3

Step 4 Step 5 Step 6

Step 7

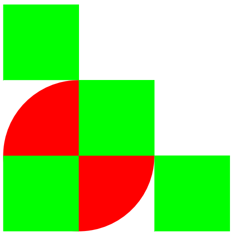
A.9. Option `overlap`

```
\setpixelartdefault{tikz={scale=.3}}

\newcommand{\three}{
  .111.
  11.11
  ...11
  ...11
  ..11.
  ...11
  ...11
  11.11
  .111.
}
\pixelart[squares={overlap=.1}]{
  \three
}
\pixelart[squares={overlap=0}]{
  \three
}
\pixelart[squares={overlap=-.1}]{
  \three
}
```



A.10. Command \tikzpixelart

```
\begin{tikzpicture}
  \fill[red] (0, 0) circle (1);
  \tikzpixelart[tikz={color=green}]{(-1, -1)}{
    1
    .1
    1.1
  }
\end{tikzpicture}
```



A.11. Pictures used for the benchmark

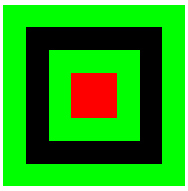
The benchmark discussed in section 5.5 (page 10) uses three files with three big `\pixelart` commands. A small version of those commands is shown below.

mono Every single pixel of the same color.

```
    \pixelart{
      KKKKKKKK
      KKKKKKKK
      KKKKKKKK
      KKKKKKKK
      KKKKKKKK
      KKKKKKKK
      KKKKKKKK
      KKKKKKKK
    }

```

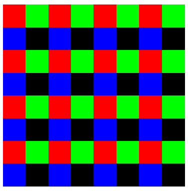
concentric Cencentric squares, that is, several area with adjacent pixels of the same color.

```
\pixelart{
  GGGGGGGG
  GKKKKKKG
  GKGGGGKG
  GKGRRGKG
  GKGRRGKG
  GKGGGGKG
  GKKKKKKG
  GGGGGGGG
}

```



checker A (kind of) checker board, where no two adjacent pixels have the same color.

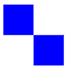
```
\pixelart{
  RGRGRGRG
  BKBKBKBK
  RGRGRGRG
  BKBKBKBK
  RGRGRGRG
  BKBKBKBK
  RGRGRGRG
  BKBKBKBK
}

```



A.12. Default options

```
\setpixelartdefault{colors=mono, tikz={scale=.2, red}}

From now on, pixel art pictures will be small, and red:
\pixelart{1. .1}.

Excepted this one:
\pixelart[
  tikz={scale=.4, blue}
]{1. .1}.

Back to normal:
\pixelart{1. .1}.
```

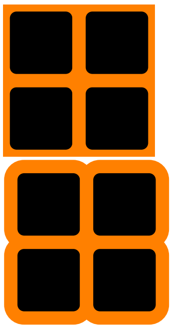
From now on, pixel art pictures will be small, and red: .

Excepted this one: .
Back to normal: .

A.13. Option `margin`

In the first picture, you can see that part of the orange line has been clipped away. In the second one, with option `margin` correctly set, it is not clipped away.

```
\setpixelartdefault{squares}
\tikzset{pixelart/.style={
  draw=orange,
  rounded corners=5,
  line width=5,
}}

\pixelart{11 11}

\pixelart[margin=.1]{11 11}
```



B. Change History

This is a raw copy of the *CHANGELOG.md* file that can be found in the git repository of pixelart.

- pixelart 1.0.2 (2023-02-20)
  - Fix error while trying to load pixelart.lua (closes #2).

  – Louis Paternault spalax@gresille.org[1]

- pixelart 1.0.1 (2023-02-18)
  - Bugs
    - Minor change to take into account backward-incompatible change in `luakeys` v0.13.0 (thanks Jonathan P. Spratte).
    - Pixelart (with option `square`) no longer fails when asked to draw an empty pixelart.
  - Documentation
    - Minor changes and improvements.

  – Louis Paternault spalax@gresille.org[2]

- pixelart 1.0.0 (2022-11-16)
  - Full, backward incompatible, rewrite of `pixelart`, in Lua.
  - Keep version 0 of `pixelart` as `pixelart0`.

---

[1] `mailto:spalax@gresille.org`

[2] `mailto:spalax@gresille.org`

– Louis Paternault spalax@gresille.org[3]

- pixelart 0.3.0 (2022-11-16)
  - Renamed `pixelart` to `pixelart0`.
  - Mark `pixelart0` as unmaintained.

– Louis Paternault spalax@gresille.org[4]

- pixelart 0.2.0 (2018-02-25)
  - Add a `draft` package option.

– Louis Paternault spalax@gresille.org[5]

- pixelart 0.1.2 (2018-01-13)
  - Fix bug: First line-break is now automatically ignored.
  - Minor documentation fixes and improvements.

– Louis Paternault spalax@gresille.org[6]

- pixelart 0.1.1 (2017-12-08)
  - Fix CTAN URLs.

– Louis Paternault spalax@gresille.org[7]

- pixelart 0.1.0 (2017-12-05)
  - First published version.

– Louis Paternault spalax@gresille.org[8]

---

[3] `mailto:spalax@gresille.org`

[4] `mailto:spalax@gresille.org`

[5] `mailto:spalax@gresille.org`

[6] `mailto:spalax@gresille.org`

[7] `mailto:spalax@gresille.org`

[8] `mailto:spalax@gresille.org`

C. Index