Package 'epsdice' — a scalable dice font

2007/02/15 – Version 2.1
Thomas Heim
(thomas.heim@unibas.ch)

1 Introduction

Dice fonts are already available in METAFONT format. (I should know, I wrote one myself: `dice3d.mf`.) For some applications it seems preferable, however, to have the fonts in scalable form. The package `epsdice` fills this gap.

`epsdice.sty` provides a single command, `\epsdice{#1}`, taking one mandatory argument, an integer from 1 to 6, and recognising one optional argument, namely `[black]` for a black die with white dots.[1] The default behavior (i.e., with *no* optional argument or with *any optional argument other* than `[black]`) results in a white die with black dots.

The mandatory argument may also be the value of a counter. Thus automatic representation of (generated or input) data is possible as well. Of course, values outside the range 1–6 will not be represented as die faces.

Depending on the value of the argument, `\epsdice` includes the appropriately clipped region from a file containing simple drawings of die faces. The drawings scale with the current font because the height of the included graphics file is set in terms of 'ex' units.

2 Requirements and limitations

`epsdice` relies on the `graphicx` package to include the drawings. It also uses the `ifthen` package to validate the arguments (error checking).

The package has been tested with two widely used package drivers for `graphicx`: `[dvips]` and `[pdftex]`. By default, the command `\epsdice` in-

[1] Thanks to Chrisoph Zurnieden for suggesting the extension with reversed colors, and for adjusting the drawings to this purpose.

cludes `epsdice.eps` under LaTeX 2_ε, and `epsdice.pdf` under pdfLaTeX, respectively[2] (unless your `graphics.cfg` file specifies non-standard settings).

Clipping is supported only in pdfTeX versions ≥ 0.14. You will also need an up-to-date version of `pdftex.def`: Clipping is not supported in versions `v0.03c [2000/09/04]` or earlier. If your pdfTeX is up-to-date but you still receive error messages about clipping not being supported, get the latest `pdftex.def` here:

`http://www.tug.org/applications/pdftex/pdftex.def`

I tested the package successfully with pdfTeX 0.14e and `pdftex.def` v0.03f `[2000/11/10]`.

3 Configuring this package

To finish the installation of the package, simply move the following files somewhere in your "local texmf tree":

`epsdice.sty, epsdice.cfg, dice.eps, dice.pdf`

e.g. to a new subdirectory `/tex/latex/epsdice/`. Refresh the TeX-system's file name database, and that's it, you're all set! The remainder of this section applies only if you want to change the default settings.

The command `\epsdice` works by including an external file, so this file must be found by LaTeX 2_ε. The package comes with a configuration file `epsdice.cfg` containing the file name in the variable `\dicefile`. By default, this variable points to the file `dice.{eps|pdf}` (without file name extension). If you don't like my drawings and would like to "roll your own", a look at section 5 may give some hints. It contains the same PostScript code as `dice.eps`, with some comments about what's going on. Conversion to PDF format has been accomplished with `epstopdf` (using GhostScript), see pdfTeX's web page.

If you intend to keep the external file with the dice drawings in a different place or if you want to use your own version you have to edit the configuration file `epsdice.cfg` accordingly, by adding the path to the file name, or by changing the file name in the variable `\dicefile`. If a configuration file does not exist, the external file is assumed to be located in the present working directory.

[2] Many thanks to Rolf Niepraschk for his help with the latter!

4 Examples

Die faces in an 11pt environment: The pictures fit into the surrounding text, as in ⚀ ⚁ ⚂ ⚃ ⚄ ⚅. The code used to set these die faces was:

```
\epsdice{1} \epsdice[black]{2} \epsdice[black]{3}
\epsdice[red]{4} \epsdice[black]{5} \epsdice[white]{6}
```

The only effective option to the command `\epsdice` is `[black]`. Anything else results in the default behavior with white background.

Here is some `\Large` text. The die pictures scale accordingly:[3] ⚂ ⚄. See?

Finally, note that the package works with standard counters, too: Here's the result of `\epsdice{\value{section}}` ⚃.

5 Drawing dice in PostScript

Something as simple as a "square with dots on it" is rather straightforward to draw in PostScript. The first line

```
%!PS-Adobe-2.0 EPSF-1.2
```

is just a standard header. Next comes the bounding box: Each face measures 32×32 pt, centered in a 43×43 pt box. For six faces in a row, this gives a bounding box of 258×43 pt. We have two rows of dice, the lower row containing the white dice with black dots, the upper one the black dice with white dots. So the total bounding box is 258×86 pt. The background color will be stored in the variable $bw = 0$ or 1.

```
%%BoundingBox: 0 0 258 86
```

The `/frame` macro defines a simple box with rounded corners. It takes one argument, the face index n, and calculates a corresponding x-offset: $x_{\text{off}} = 43(n-1)$. The point $(x_{\text{off}}, y_{\text{off}})$ then becomes the origin of the coordinate system for this particular face. This macro uses the variable bw to calculate the vertical offset $y_{\text{off}} = 43bw$. The frame consists of straight lines separated by 5 pt from the outer margin, and of four quarter circles with radius r ($r = 5$ pt for black on white and $r = 6$ pt for white on black) centered at $(10, 10)$, $(32, 10)$, $(10, 32)$, and $(32, 32)$.

[3] ... as they do in footnotes, ⚂, like this.

```
/frame {
  /n exch def         % take n off the stack, store it
  /xoffset n 1 sub 43 mul def  % xoffset = 43*(n-1)
  gsave               % save the graphics state
  newpath             % start a new path
  xoffset yoffset translate % move origin to (xoffset,yoffset)
  32 5 bw sub moveto  % go to (32,5) or (32,4) in this system
  32 10 r -90 0 arc   % SE quarter circle around (32,10)
  37 bw add 32 lineto % right line
  32 32 r 0 90 arc    % NE quarter circle around (32,32)
  10 37 bw add lineto % top line
  10 32 r 90 180 arc  % NW quarter circle around (10,32)
  5 bw add 10 lineto  % left line
  10 10 r 180 270 arc % SW quarter circle around (10,10)
  closepath           % bottom line (closes the path)
  bw 0 eq { stroke }  % either paint (lower row, bw=0)
  { fill } ifelse     % or fill (upper row, bw=1) the path
  grestore            % restore graphics state
} def
```

The dot positions are labelled within a face by (x, y)-coordinates running from $(1, 1)$ for bottom left to $(3, 3)$ for top right. The dot itself is a filled circle with radius 3.5 pt. Change the dots' positions within the frame, or their radius, as you like. The `/dot` macro takes three arguments off the stack: (i) the face index, to determine the x-offset, (ii) the x-coordinate (1, 2, or 3) on the face, and (iii) the y-coordinate (1, 2, or 3) on the face.

```
/dot {
  /y exch def         % take y-coordinate off the stack
  /x exch def         % take x-coordinate off the stack
  /n exch def         % take face index n off the stack
  /xoffset n 1 sub 43 mul def  % xoffset  = 43*(n-1)
  gsave               % save the graphics state
  newpath             % start a new path
  xoffset yoffset translate % move origin to (xoffset,yoffset)
  x 8 mul 5 add       % the dot's x-position: 8*x+5
  y 8 mul 5 add       % the dot's y-position: 8*y+5
  3.5 0 360 arc       % a circle with radius 3.5 pt
  closepath           % close the circle
  bw setgray          % set the appropriate color
  fill                % fill the circle
```

```
  grestore            % restore graphics state
} def
```

Now choose a linewidth (2 pt)

```
2 setlinewidth
```

and loop over the background color (and hence the rows) from 0 to 1. The color counter is stored in bw and used to determine y_{off} as well as the radius of the circles, $r = 5 + bw$ pt.

```
0 1 1 {               % set up loop from 0 to 1
  /bw exch def        % store loop index in bw
  /yoffset bw 43 mul def  % yoffset  = 43*bw
  /r 5 bw add def     % radius r = 5+bw
  0 setgray           % set color to black for all frames
```

All we have to do now is draw the six faces

```
  1 1 6 { frame } for % call /frame macro for n=1..6
```

and fill in the appropriate dots. The bottom left and top right dots appear on faces 2, 3, 4, 5, and 6. Thus there is an outer loop from 2 to 6 in steps of 1. For each of these face indices, we call the dot macro with (x, y)-argument equal to $(1, 1)$ and $(3, 3)$. This can be achieved with an inner loop running from 1 to 3 in steps of 2. Because the x- and y-coordinates coincide for these particular points, we obtain them by duplicating the inner loop index. The `/dot` macro gobbles not only the x- and y-coordinates but the face index as well. In order to draw *two* dots on the same face, we thus have to duplicate the outer loop index, too.

```
  2 1 6 {      % start loop 2 to 6 in steps of 1
    dup        % duplicate index (two dots on each face)
    1 2 3 {    % start loop 1 to 3 in steps of 2
      dup      % duplicate loop index -> 1 1 and 3 3
      dot      % call /dot macro, using three arguments
    } for      % inner loop (dot positions)
  } for        % outer loop (face index)
```

The center dot appears on faces 1, 3, and 5. We get it in a simple loop from 1 to 5 in steps of 2, calling the `/dot` macro for each index together with the coordinate set $(2, 2)$ corresponding to the center.

```
  1 2 5 { 2 2 dot } for  % (2,2) dot on 1, 3, 5
```

The top left and the bottom right dots appear on faces 4, 5, and 6. Again, in order to draw *two* dots on each face, we have to duplicate the loop index, i.e., the face index:

```
  4 1 6 {      % start loop 4 to 6 in steps of 1
    dup        % duplicate index (two dots on each face)
    1 3 dot    % call /dot macro for (1,3) dot on this face
    3 1 dot    % call /dot macro for (3,1) dot on this face
  } for        % loop over face index
```

Finally, on face 6 we have two dots in positions $(1, 2)$ and $(3, 2)$. We can use a loop, but then we have to swap the loop index, i.e., the dot's x-coordinate, with the (constant) face index 6 inside the loop.

```
  1 2 3 {      % start loop 1 to 3 in steps of 2
    6 exch     % swap loop index and constant face index
    2          % push constant y-coordinate on stack
    dot        % call /dot macro for (x,2) dot on face 6
  } for        % loop over x-coordinates 1 and 3
```

And this closes the loop over background colors:

```
} for          % loop over bw 0 and 1
```

That's it!

```
%%EOF
```

6 Errors

Of course it doesn't make sense to ask for die faces with arguments other than 1, 2, 3, 4, 5, or 6. The package generates error messages and simply prints the offending argument in the TEX output.