A	B	C
1	2	3
4	5	6

A	B	C
1	2	3
4	5	6

shapepar.sty

Donald Arseneau
Vancouver, Canada
asnd@triumf.ca

v 2.0 Dec 2002

`\shapepar`: a macro to typeset paragraphs in a specific shape. The size is adjusted automatically so that the entire shape is filled with text. This package is for Plain TEX, LATEX, or similar.

1 \shapepar and \Shapepar

The `\shapepar` macro (or 'command') is used to typeset paragraphs of a specified shape, where the total size is adjusted automatically so that the entire shape is filled with text, and the shape may include separate pieces and holes. This is distinct from the normal `\parshape` command which specifies a simple shape *and* a size that may be partially filled, or over-filled, from top to bottom. In a `\shapepar` there can be no displayed math, and no '`\vadjust`' material, (including `\vspace`). This style is mainly intended for cards, invitations etc., not for whole books! Although short paragraphs process much faster, only long paragraphs accurately fill complex shapes.

These macros work for both LATEX and plain TEX. For LATEX, specify `\usepackage{shapepar}`, or for Plain, `\input shapepar.sty`.

The command `\shapepar` should be used at the beginning of a paragraph, and it applies to the entire paragraph. There is one optional length parameter: a fixed scale, ⟨*scale_len*⟩; and one required parameter: a description of the shape, ⟨*shape_spec*⟩.

`\shapepar [`⟨*scale_len*⟩`]{`⟨*shape_spec*⟩`}` Text of the paragraph

The text of the paragraph is delimited by a blank line or `\par`, but is not literally a *parameter*, so verbatim macros will work there. If you want to typeset two paragraphs in one shape, then use `\endgraf` or `\\` to split them.

Ordinarily, the scale is calculated automatically so the pargaraph best fills the shape. If a scale length is given, then the shape is reproduced so one unit of the ⟨*shape_spec*⟩ equals the ⟨*scale_len*⟩, and the shape is filled with white space after the paragraph text. The ⟨*scale_len*⟩ is much like `\unitlength` for the picture environment. For an application of a fixed scale, see the `\CDlabel` macro.

With the `\shapepar` command, the text will be typeset centered on the page using the specified shape (specifically, the shape's ⟨*h_center*⟩ will be centered on the page; see below). A `\shapepar` should not break across pages (due to inter-line penalties) but that feature is not guaranteed.

The `\Shapepar` macro (capital S) typesets the shaped paragraph in a box (`\vtop` or `\parbox[t]`) without extra horizontal padding. If it occurs in vertical mode, special care is taken with the line-spacing around this box, but the line-spacing might be better with plain `\shapepar`. `\Shapepar` is particularly useful for `\fbox` or `\put`.

2 \cutout

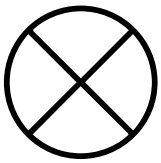
A shaped paragraph can be incorporated with the page in a third way: nestled in a cut-out at the side of the running text.

`\cutout {`⟨*side*⟩`} (`⟨*h_offset*⟩`,`⟨*v_offset*⟩`)` ⟨*settings*⟩ `\shapepar` ...

The ⟨*side*⟩ argument is required, and must be 'l' or 'r', indicating which margin (left or right) the shape should occupy. By default (in the absence of other parameters) the shaped paragraph will be placed so its center-line (⟨*h_center*⟩) is at the specified edge of the running text, and its first line is level with the first line of the ensuing text. This position may be changed by specifying the optional horizontal and vertical offset distances `(`⟨*h_offset*⟩`,`⟨*v_offset*⟩`)` in parentheses. A positive ⟨*h_offset*⟩ moves the shape further right, and a positive ⟨*v_offset*⟩ moves it down. A negative ⟨*v_offset*⟩ will move the shape upwards, where it might overlap preceding text, which will *not* be cut away to accommodate.

Yes, the text coming after the `\cutout \shapepar` is cut out (surprise!) to fit the shape, leaving a gap of '`\cutoutsep`'. This length (dimen) parameter is initialized to 12 pt; you may set it as you please. The cutout separation is applied by looking at the paragraph shape expanded by `\cutoutsep` in all directions, to give the same gap at every slope. The cutout only lasts for one paragraph.[1] To extend the effect further, divide paragraphs with '`\\[\parskip] \indent`' (LaTeX) or '`\hfill \break \indent`' so TeX does not treat them as separate paragraphs. Furthermore, the cut-out paragraph should not end in a local group, so make sure there is an explicit `\par` or blank line there, outside of braces.

The combination of `\cutout` with fixed scale allows an entirely different application: producing cutouts for graphics. First, produce a rough ⟨*shape_spec*⟩ for the image to delineate its left and right borders (you can ignore all internal detail) or use an appropriate pre-defined shape. Then use `\includegraphics` as the text of the shaped paragraph. There is a difficulty though: we want the image to take the place of the entire paragraph, not to stand up on the



[1] Re-defining `\par` for the extent of the cutout is an attractive feature, but has conflicts with various TeX formats. It seems best to leave the paragraph control in the hands of the user. This will likely change in the future.

top line. There are a number of methods for lowering the graphic appropriately (and some methods that will not work with `\shapepar`); three that work are:

```
\hfill \makebox [0pt][c]{\raisebox{1ex-\height} % requires calc.sty
    {\includegraphics{⟨file⟩}}}
```
or
```
\hfill
\begin{minipage}[t]{0pt}
\vspace {-1ex} % \vspace works here only
\centerline{ \includegraphics{⟨file⟩} }
\end{minipage}%
\hfill
```
and, most general,
```
\begin{picture}(0,0)
\put (x, y){\includegraphics{⟨file⟩}}
\end{picture}
```

The first two adjust the height automatically, but assume the top line of the shape is horizontally centered. The picture environment handles positioning more flexibly, but requires you to provide parameters x, y. Just try approximate values, make a test run, measure the offset, and correct. This should only require one trial, not an endless cycle. Because the reference point for the shaped paragraph is based on the *text* it contains, and not just the specified shape, some adjustment will usually be needed to position a shaped paragraph precisely.

3 Shapes

There are some shapes predefined in shapepar.sty (square, circle, circle-with-hole, diamond, heart, star and hex-nut) which are used as examples in the instructions below. Each of these shapes is stored in a macro, and there is a command to use that shape:

`\squareshape`	`\squarepar`	Square
`\circleshape`	`\circlepar`	Circle
`\CDshape`	`\CDlabel`	Circle with circular hole (`\CDlabel` uses a fixed scale to fit a compact disc)
`\diamondshape`	`\diamondpar`	Rhomboid 'diamond' (♢)
`\heartshape`	`\heartpar`	Heart (symbolic shape ♡)
`\starshape`	`\starpar`	Five-point star
`\nutshape`	`\nutpar`	Nut for bolt (hexagon with circular hole)

For example, `\heartpar{`⟨*text*⟩`}` performs `\shapepar {\heartshape}` ⟨*text*⟩`\ \ $\heartsuit$\par` (ending the text with a heart symbol).

More shape definitions are provided in separate files named ∗shape.def. Look on your disk to be sure, but the list should include:

candleshape	A burning candle
TeXshape	The TEX logo
Canflagshape	The Canadian flag

Please contribute your shapes!

Although defining shapes by hand can be difficult, there are programs to aid you.

3.1 proshap.py by Manuel Gutierrez Algaba

proshap.py (ver 1.1) is a python script written by Manuel Gutierrez Algaba to produce shape definitions from rough 'ascii art'. There is no instruction manual, so here are Donald Arseneau's observations. There is not much of a user interface; look in proshap.py (which is a plain text file) and see how the various 'test' shapes are defined (note the triple-double quotes). Choose one of them, or add a new one, then change the line 'test = test3' to select the desired picture. Execute 'python proshap.py' which will output a definition of `\bassshape` to the screen and to the file 'result.tex'. The goulish face you see here is the test3 shape. You should be aware that the characters in the ascii input are treated as square, even though they are taller than they are wide, so the output shape specification will be taller and thinner than the input text. There also seems to be a problem with all 'bottoms': flat bottoms of text blocks and of holes are expanded downwards to end at a point. Compare this face to the original face in proshap.py. Warning: These instructions and observations are probably wrong; the author does not program in python so can't even read the code properly. For now, look for proshap.py bundled with shapepar.sty.

3.2 proshap.py by Manuel Gutierrez Algaba

aarati (ver 1.1) is a python script written by Manuel Gutierrez Algaba to produce shape definitions from rough 'ascii art'. There is no instruction manual, so here are Donald Arseneau's observations. There is not much of a user interface; look in proshap.py (which is a plain text file) and see how the various 'test' shapes are defined (note the triple-double quotes). Choose one of them, or add a new one, then change the line 'test = test3' to select the desired picture. Execute 'python proshap.py' which will output a definition of `\bassshape` to the screen and to the file 'result.tex'. The goulish face you see here is the test3 shape. You should be aware that the characters in the ascii input are treated as square, even though they are taller than they are wide, so the output shape specification will be taller and thinner than the input text. There also seems to be a problem with all 'bottoms': flat bottoms of text blocks and of holes are expanded downwards to end at a point. Compare this face to the original face in proshap.py. Warning: These instructions and observations are probably wrong; the author does not program in python so can't even read the code properly. For now, look for proshap.py bundled with shapepar.sty.

aarati (ver 1.1) is a python script written by Manuel Gutierrez Algaba to produce shape definitions from rough 'ascii art'. There is no instruction manual, so here are Donald Arseneau's observations. There is not much of a user interface; look in proshap.py (which is a plain text file) and see how the various 'test' shapes are defined (note the triple-double quotes). Choose one of them, or add a new one, then change the line 'test = test3' to select the desired picture. Execute 'python proshap.py' which will output a definition of `\bassshape` to the screen and to the file 'result.tex'. The goulish face you see here is the test3 shape. You should be aware that the characters in the ascii input are treated as square, even though they are taller than they are wide, so the output shape specification will be taller and thinner than the input text. There also seems to be a problem with all 'bottoms': flat bottoms of text blocks and of holes are expanded downwards to end at a point. Compare this face to the original face in proshap.py. Warning: These instructions and observations are probably wrong; the author does not program in python so can't even read the code properly. For now, look for proshap.py bundled with shapepar.sty.

3.3 ShapePatch by Christian Gollwitzer

ShapePatch is an amazing utility written by Christian Gollwitzer. It allows you to simply draw the shape you want with Xfig, and then convert it to a shapepar shape definition, either by manually running fig2dev (a component of transfig), or by choosing 'Export/Shape' in Xfig. Its odd name is indicative of its implementation: it is an upgrade or 'patch' to both the Xfig program and the transfig tool-set. ShapePatch can be found on CTAN under `graphics/trans` `fig-shapepatch`, and includes instructions (README) and sample shapes besides the patch itself. This smiley face is one of the sample shapes. In the future, the ShapePatch utility will be included in transfig, so there will be no need to install the patch. Besides enabling a graphical utility (Xfig) to produce shapepar specifications, the ShapePatch upgrade opens up great libraries of existing Fig clip-art to be used for paragraph shapes. For example, a Canadian flag taken from the Xfig flag library is one of the examples provided with ShapePatch, for comparison with the hand-coded one bundled with shapepar.sty. (I prefer my own version because there is an error in the maple leaf shape on the Xfig flag library version. The error is not the fault of ShapePatch, but ShapePatch faithfully reproduces the faulty shape.)

4 Shape Syntax

The syntax rules for ⟨*shape_spec*⟩ are very specific, and must be followed closely. In these rules, `{ }` mean explicit braces, [ ] denote optional parts, ⟨ ⟩ surround a keyword that is defined (perhaps loosely), and | means 'or'; do not type [ ] ⟨ ⟩ or |, but do type `{ }`.

⟨*shape_spec*⟩ = `{`⟨*h_center*⟩`}` ⟨*lines*⟩

⟨*lines*⟩ = ⟨*line_spec*⟩ [\\⟨*lines*⟩]

That is, the shape is specified as a single number in braces, followed by the specifications for the lines, with the lines separated by \\. The resulting paragraph will have its ⟨*h_center*⟩ position centered on the page, or used as a reference point. It is a number like 10.5, without explicit units, but using

the same length scale as the lengths and positions in the ⟨*lines*⟩. Ordinarily, shapepar will determine the unit length that best fits the text, but will use a fixed scale when specified for `\shapepar`.

The lines in the spec are not lines of text, nor are they the lines that you would use to draw the shape itself. They are horizontal scans across the shape at regular or irregular intervals. Complex curved shapes need many scan lines for accurate rendering, while simple shapes need only a few. To determine the line specifications, start by drawing the shape on paper, then draw a series of horizontal lines across it, including lines that just touch the top and the bottom of the figure, and, preferably, lines through each sharp corner. Each line crosses over pieces of the figure in some region. These intersections of line and figure define a ⟨*line_spec*⟩.

⟨*line_spec*⟩ = `{`⟨*v_pos*⟩`}` ⟨*segment*⟩ [other ⟨*segment*⟩s]

The ⟨*v_pos*⟩ is the vertical position of the line, increasing from top to bottom. Each ⟨*line_spec*⟩ usually has a position greater than or equal to that of the previous line, and with all ⟨*v_pos*⟩ > -1000. The exception is that between consecutive lines relating to completely disconnected parts of the figure the ⟨*v_pos*⟩ may decrease (backspacing). This allows text to flow from one disconnected area to another in sequence (see the Canadian flag shape). Each ⟨*segment*⟩ represents a region where text will go in the final paragraph; it is the segment of the horizontal scan line that overlaps the body of the figure. There are five types of segment:

⟨*segment*⟩ = `t{`⟨*pos*⟩`}{`⟨*len*⟩`}` | `b{`⟨*pos*⟩`}` | `e{`⟨*pos*⟩`}` | `s` | `j`

`b{`⟨*pos*⟩`}`	begin text block at a point at horizontal position ⟨*pos*⟩
`e{`⟨*pos*⟩`}`	end text at a point at horizontal position ⟨*pos*⟩
`t{`⟨*pos*⟩`}{`⟨*len*⟩`}`	text segment at position ⟨*pos*⟩ with length ⟨*len*⟩
`s`	split text block (begin a gap)
`j`	join two text blocks (end a gap)

The most common type of segment is `t` (text). The other types are degenerate in that they are single points rather than finite segments. Types `s` and `j` have no explicit position, but they must appear between text segments, and those texts should abut; e.g., `t{3}{2}st{5}{4}` (text from 3 to 5 and text from 5 to 9).

Let's jump right into a simple example, and the meanings will be clearer. A rhombus 'diamond' shape can have the four vertices, with coordinates shown in figure 1. This shape can be exactly specified by just three scan lines passing through the vertices. The intersections of the scan lines with the shape's edges occurs at the vertices and so the shape specification is:

`{3}`	⟨*h_center*⟩: $x = 3$
`{0}b{3}\\`	text block begins at point $y = 0, x = 3$
`{4}t{0}{6}\\`	scan (at $y = 4$) crosses text (len 6) starting at $x = 0$
`{8}e{3}`	text block ends at point $y = 8, x = 3$

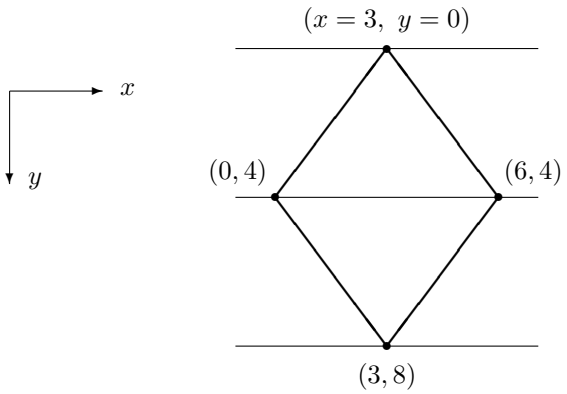


Figure 1: Diamond shape, showing vertex locations and scan lines.

Other specification lines, such as

```
{6}t{1.5}{3}\\
```

could be inserted, but would make no difference – the shape is interpolated linearly between scan lines.

Every block of text must start with a `b` specifier and end with an `e` spec on some line below. Every segment specified by `t` must have a length greater than zero. If two blocks of text merge to form one (like at the notch of a heart shape) there should be a `j` spec at the point of junction. If one block bifurcates or splits (like at the top of a hole in a doughnut) there should be an `s` spec.

Thus, the first line for any valid shape description must consist of only `b` segment descriptors; the last line can only have `e` type descriptors. Although the definition of the units is arbitrary, the numbers should range in magnitude from ~ 0.1 to 100 to avoid numeric overflows and underflows.

If there are errors in the format of the specification, `\shapepar` might complain with the error message

Shaped Paragraph Error: Error in specification. Check carefully!

At this point you may as well type `x` or `e`, to exit from TEX, as there is very little chance that TEX will continue successfully. You might also get one of TEX's regular error messages, like

Illegal unit of measure (pt inserted).

or

Missing number, treated as zero.

or you might get no error message at all, just ridiculous formatting. Check shape syntax carefully against the rules and the examples before running them through TEX.

```
\newcommand\heartshape{
{20}{0} b{13.32}          b{26.68}
\\{.14} t{10.12}{4.42}    t{25.46}{4.42}
\\{.7}  t{9.14}{7.16}     t{23.7}{7.16}
\\{1.4} t{8.4}{9.02}      t{22.58}{9.02}
\\{2.1} t{7.82}{10.42}    t{21.76}{10.42}
\\{2.8} t{7.36}{11.58}    t{21.06}{11.58}
\\{3.5} t{6.98}{12.56}    t{20.46}{12.56}
\\{4.2} t{6.68}{13.32} j  t{20}{13.32}
\\{4.9}   t{6.48}{27.04}
\\{5.6}   t{6.34}{27.32}
\\{6.3}   t{6.28}{27.44}
\\{7}     t{6.26}{27.48}
\\{7.7}   t{6.27}{27.46}
\\{8.4}   t{6.32}{27.36}
\\{9.1}   t{6.4}{27.2}
\\{9.8}   t{6.52}{26.96}
\\{10.5}  t{6.68}{26.64}
\\{11.9}  t{7.12}{25.76}
\\{13.3}  t{7.72}{24.56}
\\{14.7}  t{8.51}{22.98}
\\{16.1}  t{9.5}{21}
\\{17.5}  t{10.69}{18.62}
\\{18.9}  t{12.08}{15.84}
\\{20.3}  t{13.7}{12.6}
\\{21.7}  t{15.62}{8.76}
\\{22.4}  t{16.7}{6.6}
\\{23.1}  t{17.87}{4.26}
\\{24.6}  e{20}
}
```

In faith, I do not love thee with mine eyes, For they in thee a thousand errors note, But is my heart that loves what they despise, Who in despite of view is pleased to dote; Nor are mine ears with thy tongue's tune delighted, Nor tender feeling to base touches prone, Nor taste nor smell, desire to be invited To any sensual feast with thee alone; But my five wits nor my five senses can Dissuade one foolish heart from serving thee, Who leaves unswayed the likeness of a man, Thy proud heart's slave and vassal wretch to be; Only my plague thus far I count my gain, That she that makes me sin awards me pain.

♡

Figure 2: Specification for the heart shape, and an example.

What to do if the figure does not start at a point – if it has a flat top? It can start at a single point, but have the next scan line at the same vertical position! A square paragraph is specified by:

`{1}`	centerline is at $x = 1$
`{0}b{0}\\`	begin at $(0, 0)$
`{0}t{0}{2}\\`	text at $y = 0$, width $= 2$
`{2}t{0}{2}\\`	text at $y = 2$, width $= 2$
`{2}e{1}`	end at $(1, 2)$

Now let's get more ambitious. A heart shape must have two simultaneous beginnings, a short stretch where there are two separated text areas ending with a join, whereafter there is just one block of text leading to the final bottom point. Figure 2 shows the heart-shape specification. Find the two `b` specifiers at the beginning, and find the `j` a few lines below; notice that above the `j` there are two segments per line, but only one below it – the two lobes join at the `j` point: 20. I drew this heart freehand, and measured lengths from the sketch, so you should be able to do better! The spec has many scan lines so that the smooth curves are preserved, but there are probably more lines than necessary.

Text shapes can have holes. For example, a doughnut-shape would have a `b` on the first line, followed by some lines with a single `t`, then a line with `t s t` at the start of the hole. The hole is represented by lines with two `t` specs – the gap between them is the hole. A line with `t j t` ends the hole. There are more lines with single `t`, and then an `e` line to end with. Such a doughnut is used by the `\CDlabel` shape, but the example given in Figure 3 is a nut. Not a doughnut, but a hex-nut (for a machine screw or bolt) – a regular hexagon with a circular hole in the center. The hexagon is flat on top and bottom so the specification begins and ends like the square shape. The circle is rendered as a 24-gon, beginning with a split (`s`) of the surrounding text and ending with a join (`j`). If the spacing of the scan lines looks odd, it is because the hexagon alone would need just 5 scan lines (at only 3 distinct locations), but the circle needs many; the points on the circle are at 15-degree intervals.

```
\newcommand\nutshape{
{0}
{0}     b{0}\\
{0}     t{-12.5}{25}\\
{11.65} t{-19.23}{19.23} s t{0}{19.23}\\
{11.99} t{-19.42}{16.835}  t{2.59}{16.835}\\
{12.99} t{-20}{15}         t{5}{15}\\
{14.58} t{-20.92}{13.85}   t{7.07}{13.85}\\
{16.65} t{-22.11}{13.45}   t{8.66}{13.45}\\
{19.06} t{-23.51}{13.85}   t{9.66}{13.85}\\
{21.65} t{-25}{15}         t{10}{15}\\
{24.24} t{-23.51}{13.85}   t{9.66}{13.85}\\
{26.65} t{-22.11}{13.45}   t{8.66}{13.45}\\
{28.72} t{-20.92}{13.85}   t{7.07}{13.85}\\
{30.31} t{-20}{15}         t{5}{15}\\
{31.31} t{-19.42}{16.835}  t{2.59}{16.835}\\
{31.65} t{-19.23}{19.23} j t{0}{19.23}\\
{43.3}  t{-12.5}{25}\\
{43.3}  e{0}
}
```

Figure 3: Specification for the nut shape. (The definition in shapepar.sty is the same except that the spaces are removed.)

5 Configuration

There are several parameters that control details of how `\shapepar` functions, varying from those (one) that will be set often, to some that are cryptic, and require editing shapepar.sty itself. Here we will explain some of those operational details and the parameters that control them.

5.1 Cut-out separation

The cutout separation is applied by looking at the paragraph shape expanded by `\cutoutsep` in 'all' directions, to give the same gap at every slope. That is a white lie. The shape's '`\parshape`' is actually regenerated at a finer line-spacing (but the same scale factor as for the shaped paragraph), and the leftmost (or rightmost) position of each finer-parshape line is used to exclude an octagon (as an approximation to a circle) centered on that position. The octagon is not even regular, but is extended vertically by `\cutoutsepstretch` × `\baselineskip` to allow for the height and depth of characters. Figure 4 illustrates these parameters.

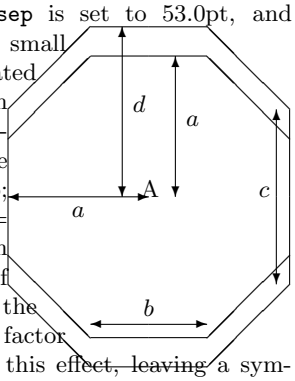


Figure 4: This is an example with a tiny (unrecognizable) square shaped 'paragraph' and a large cutout separation. Here `\cutoutsep` is set to 53.0pt, and `\cutoutsepstretch` is 1.0. Since the 'paragraph' is so small ('A'), and the separation so great, the cutout is dominated by the octagonal expansion of the shape which is drawn for `\cutoutsep` alone and for the full exclusion zone including `\cutoutsepstretch`. The labelled distances are as follows: a = `\cutoutsep`; b = 0.828 `\cutoutsep`; $c = b + 2 \times$ `\cutoutsepstretch` × `\baselineskip`; $d =$ $a +$ `\cutoutsepstretch` × `\baselineskip`. The exclusion zone (outer octagon) applies to the baseline position of each paragraph line, so the characters do intrude into the octagon from above and below. The vertical expansion factor `\cutoutsepstretch` should be chosen to just counteract this effect, leaving a symmetric border of white space (inner octagon). In ordinary use, with a larger shaped paragraph and a smaller separation, the expansion of the shape should appear equal in all directions (circular); it is only in this contrived example that it is revealed as an octagon.

A good place to set these parameters is between `\cutout` and `\shapepar` so they apply locally to just that instance.

`\cutoutsep` Distance separating shaped paragraph and the surrounding cutout text.

Type: length (dimen register)
Default: 12 pt
Set with: `\setlength`
Example: `\setlength {\cutoutsep }{1cm}` or `\cutoutsep = 5pt`

`\cutoutsepstretch` Vertical extension of cutout gap, given as a fraction of `\baselineskip`, which accounts for the height and depth of characters. Note that, even with a good setting for `\cutoutsepstretch`, the separation above and below the shaped paragraph may not match the side spacing simply because the cutout text has rigid baseline skips.

Type: macro ('command', but really 'data')
Default: .5
Set with: `\renewcommand` or `\def`
Example: `\renewcommand {\cutoutsepstretch }{.75}`

\RefineBaselines Fineness of cut-out matching to lines (number of reference points per line of cut-out text).
Type: integer constant (not a LaTeX counter)
Default: 3
Set with: `\renewcommand`, `\def`, `\chardef`, `\mathchardef` etc.
Example: `\renewcommand {\RefineBaselines }{2}` or
`\chardef \RefineBaselines = 4`

5.2 Scale length optimization

When `\shapepar` is used without an explicit scale length, it must determine a scale that allows the given text to fill the shape. It makes a first guess by comparing the length of the text with the area of the shape specification, and then typesets the paragraph at that scale. If the text does not fit well, then `\shapepar` changes the scale and tries again. It will make as many as `\ScaleMaxTries` trial paragraphs:

\ScaleMaxTries How many times will `\shapepar` try to get the size of the paragraph?
Type: integer constant (not a LaTeX counter)
Default: 9
Set with: `\renewcommand`, `\def`, `\chardef`, `\mathchardef` etc.
Example: `\renewcommand {\ScaleMaxTries }{7}` or
`\chardef \ScaleMaxTries = 5`

It is quite common that text will not esaily fit a shape even when the best scaling is chosen, so it is counter-productive to set a very high value for `\ScaleMaxTries`.

There are many other numbers explicitly coded into shapepar.sty that control the scale-optimization process: choosing the initial guess, choosing the initial step size ('dscale'), changing dscale by different factors ('fac'), fitting the text to the paragraph shape, and more. These are accompanied by the comment 'optimize', and they can be changed by editing shapepar.sty.

5.3 Applying the shape

`\shapepar` cheats a bit when the horizontal gap between two bits of text is small (like down in the notch of `\heartpar`). When the gap is less than an interword space it is eliminated, and the texts are joined; when it is somewhat larger it is expanded to give it more visibility. Likewise, when a segment of text is too small, it is eliminated. There are three parameters to control this behavior.

Although they are lengths, they are all macros (commands) rather than dimen registers in order to use font-size-based length units (em).

\SmallestGap Smallest gap size allowed; smaller gaps will be eliminated (text joined).
Type: macro (command) giving a length
Default: .4 em
Set with: `\renewcommand` or `\def`
Example: `\renewcommand {\SmallestGap }{.5em}`

\SmallGap Small gap size; smaller gaps will be enlarged.
Type: macro (command) giving a length
Default: 1 em
Set with: `\renewcommand` or `\def`
Example: `\renewcommand {\SmallGap }{2em}`

\SmallestSegment Smallest segment allowed; smaller will be omitted.
Type: macro (command) giving a length
Default: .2 em
Set with: `\renewcommand` or `\def`
Example: `\renewcommand {\SmallestSegment }{10pt}`

5.4 Feedback

Since the processing is slow, some messages will be displayed to show how things are going. LaTeX users can disable this feedback by loading shapepar with `\usepackage[quiet]{shapepar}`, or they can get more verbose messages by requesting `\usepackage[noisy]{shapepar}`. There are even more verbose messages that can be activated by removing the % that hides them, but they are only useful for debugging shapepar.sty itself.

6 ~~Deficiencies~~ Future Improvements

- The shape is followed by the text baselines, and this makes the shaped paragraph marginally taller; mis-fitting of baselines makes the paragraph squatter.
- Exact placement of CD-labels on label stock and of graphics in a cut-out requires a box reference-point defined by the shape alone, not the text. But portions of the text spill out of the shape and might go off the label. How to handle best?
- The restriction to a single paragraph is a pain, especially for fixed-scale shapes. The pre-boxing and un-boxing of the text forces a single paragraph without displays. Omitting pre-boxing is easiest with fixed-scale,

but even then there may be multiple typesettings. Global assignments will accumulate and my register re-use will cause conflicts.

- The restrictions in cut-out text are likewise a problem. Should upgrade wrapfig.sty and merge some code.
- Discrete segments and fixed-scale suggest an application to magazine or poster layout. For this to be useful, excess text should be able to overflow onto the next page, and complex text (multiple paragraphs, lists, and displays) must be accepted.