The rubikrotation package

RWD Nickalls (dick@nickalls.org)
A Syropoulos (asyropoulos@yahoo.com)

This file describes version 2.0, last revised 2014/01/20

Abstract

The rubikrotation package is an extension for the rubikcube package. It provides the `\RubikRotation` command which processes on-the-fly a sequence of Rubik rotation moves (using the Perl script `rubikrotation.pl`) and returns the new Rubik cube state. It implements some basic checking of the Rubik state, and also offers a command for displaying any errors (`\ShowRubikErrors`). This package requires access to the TeX `write18` facility by using the `--shell-escape` commandline switch. The rubikrotation package has been road-tested on a Microsoft platform (with MikTeX and Strawberry Perl), on a Linux platform (TeXLive and Mandriva), and on OpenIndiana (a Solaris platform).

Contents

1 Introduction

The rubikrotation package is a dynamic extension to the RUBIKCUBE package. It consists of a style option (`rubikrotation.sty`) and a Perl script (`rubikrotation.pl`).

The primary role of the rubikrotation package is to implement on-the-fly Rubik rotation sequences using the `\RubikRotation` command. Consequently, this package requires the use of the `--shell-escape` switch to allow commandline control of the Perl script, which is really the 'engine' of this package. The rubikrotation package has been road-tested on a Microsoft platform (with MikTeX and Strawberry Perl[1]) and on a Linux platform (TEXLive and Mandriva).

The following commands are made available by `rubikrotation.sty`.

```
\RubikRotation{}
\SaveRubikState
\CheckRubikState
\ShowRubikErrors
```

Note that if the Perl script is not located in the local working directory, then some care is needed regarding placing it where your system can both find it and also run it. In this case, the setting up of a simple one or two-line configuration file may be useful or even necessary, depending on your system (see section on 'installation' below).

2 Requirements

The rubikrotation package requires the TikZ and the RUBIKCUBE packages.

[1]'Strawberry Perl' (`http://strawberryperl.com`) is a free Perl environment for MS Windows, designed to be as close as possible to the Perl environment of Unix/Linux systems.

3 Installation

3.1 Generating the files

Place the file `rubikrotation.zip` into a temporary directory, and unzip it. This will generate the following files:

```
rubikrotation.ins
rubikrotation.dtx
rubikrotation.sty
rubikrotation.pl
rubikrotation.PDF
example-rot1.tex
example-rot1.PDF
example-rot2.tex
example-rot2.PDF
```

The style option `rubikrotation.sty` is generated by running (pdf)LaTeX on the file `rubikrotation.ins` as follows:

```
  pdflatex  rubikrotation.ins
```

The documentation file (`rubikrotation.pdf`) was generated using the following steps:

```
pdflatex  --shell-escape  rubikrotation.dtx
pdflatex  --shell-escape  rubikrotation.dtx
makeindex -s gind.ist  rubikrotation
makeindex -s gglo.ist -o rubikrotation.gls  rubikrotation.glo
pdflatex  --shell-escape  rubikrotation.dtx
```

3.2 Placing the files

Place the files either in the local working directory, or where your system will find them. For a Linux system with a standard TEX Directory Structure (TDS), then:

*.sty → `/usr/local/texlive/texmf-local/tex/latex/local/rubikrotation/`
*.cfg → `/usr/local/texlive/texmf-local/tex/latex/local/rubikrotation/`
*.pdf → `/usr/local/texlive/texmf-local/doc/rubikrotation/`
*.pl → `/usr/local/bin/`

Finally, (depending on your system) update the TEX file database using the `texhash` command.

3.3 Configuration file

A plane text configuration file with the name `rubikrotation.cfg` (if one exists) will be read by the system. Such a file allows the user to adjust (a) the filename of the Perl script (`rubikrotation.pl`) and (b) the commandline code used by `rubikrotation.sty` for calling the Perl script. This sort of fine-tuning is sometimes convenient, and even necessary (depending on your system), for locating and running the Perl script. For example, on some systems it maybe preferable to use a different PATH, file name and/or a different commandline code to call the script.

`\rubikperlname` `\rubikperlcmd` The configuration file is essentially a convenient software vehicle for feeding some additional LaTeX code to the style option, and hence allow the contents of some commands to be adjusted and/or fine-tuned. For the rubikrotation package there are two particular commands we may wish to adjust. The first is that defining the filename of the Perl script, namely `\rubikperlname`. The second is that defining the commandline call, namely `\rubikperlcmd`. The default definitions for these (detailed in Section 8.2) are as follows:

```
\newcommand{\rubikperlname}{rubikrotation.pl}
\newcommand{\rubikperlcmd}{perl \rubikperlname}
```

For example, we might wish to test out a slightly modified Perl script, say with the name `rubikrotationV3.pl`. In this case we simply create, in the local working directory, a plane text configuration file (called `rubikrotation.cfg`) which includes the following line:

```
\renewcommand{\rubikperlname}{rubikrotationV3.pl}
```

Alternatively, say, if the Perl script is being installed on a Linux platform, then it would be standard to install it in the directory `/usr/local/bin`. A convenient approach in this case, therefore, would be to indicate this new PATH by including the following line in the configuration file:

```
\renewcommand{\rubikperlname}{/usr/local/bin/rubikrotation.pl}
```

and, since this would then be an essentially permanent feature, we would then place the configuration file with the style option in the `texmf-local/...` directory as described in section 3.2 (placing files). Note that in this case, we have *not* made the Perl script 'executable', since the default commandline code is effectively `perl rubikrotation.pl`, and this works just as it is.

However, if the Perl script *is* made 'executable', then a different commandline code will have to be used instead. For example, suppose the Perl script is made executable and renamed to just `rubikrotation` (ie with no filename extension), then we now have to omit the 'perl' from the default commandline. Consequently, we now need to make two command changes, which we implement by including the following two lines in the configuration file:

```
\renewcommand{\rubikperlname}{/usr/local/bin/rubikrotation}
\renewcommand{\rubikperlcmd}{\rubikperlname}
```

4 Usage

Load the packages `rubikcube.sty` and `rubikrotation.sty` in the TeX file *after* loading the TikZ package, as follows:

```
\usepackage{tikz}
\usepackage{rubikcube,rubikrotation}
```

and run (pdf)LaTeX using the `--shell-escape` commandline switch (see following section).

4.1 Enabling the TeX ‘shell’ facility

In order to enable the TeX ‘write18’ facility (so it can run the Perl script) we will need to invoke (pdf)LaTeX using the `--shell-escape` switch; say, as follows.

```
pdflatex  --shell-escape   filename.tex
```

In practice, it is probably most convenient to run this via a bash/batch file—something like the following, both of which automatically show the graphic output:

```
pdflatex  --shell-escape   filename.tex
xpdf filename.pdf
```

or

```
latex  --shell-escape   filename.tex
dvips filename.dvi
gv filename.ps
```

4.2 Test files

Two example tex files (which demonstrate the use of the package commands) are included in the package, namely:

```
example-rot1.tex   (shows 5 worked examples)
example-rot2.tex   (is a ‘test’ file for experimenting with different commands)
```

These need to be run using `--shell-escape` switch; for example:

```
 pdflatex  --shell-escape  example-rot1.tex
```

If the files give unexpected results, check-out the log file to see if the system has had any difficulties finding files etc.

5 Commands

The *only* commands which *must* be used inside a TikZ picture environment are the `\Draw...` commands (these are all provided by the RUBIKCUBE package), although most commands can be placed inside a TikZ environment if necessary. However, using commands outside the environment generally offers maximum flexibility, since the effects of commands used inside a TikZ picture environment are ‘local’ to that picture environment, and are not therefore accessable outside the environment.

The only command which must *not* be inside a TikZ environment is the `\ShowRubikErrors` command (see the notes on this command below).

5.1 \RubikRotation command

\RubikRotation The `\RubikRotation{⟨comma separated sequence⟩}` command processes a comma separated sequence of rotations, and returns the final state. For example, if we wanted to see the effect of the rotations **R, R, L, U, D** on a solved Rubik cube, we could use the following commands.

```
\begin{tikzpicture}[scale=0.7]
   \RubikCubeSolved
   \RubikRotation{R2,L,U,D}
   \DrawRubikCubeRU
\end{tikzpicture}
```

The `\RubikRotation` command results in LaTeX first writing the current Rubik state to a text file (`rubikstate.dat`), and then calling the Perl program `rubikrotation.pl`. The Perl program then reads the current rubik state from the (`rubikstate.dat`) file, performs all the rotations, and then writes the new rubik state (and any error messages) to the file `rubikstateNEW.dat`, which is then input on-the-fly by the LaTeX file and used to generate some graphic image of the cube.

Note that the `\RubikRotation` command can be either inside or before the Tikz picture environment. In fact only the '`\Draw...`' commands (from the RUBIKCUBE package) actually need to be inside a TikZ picture environment. Consequently this makes for great flexibility.

5.1.1 Arguments prefixed with a *

If any of the comma separated arguments (strings) is prefixed with a * it is not processed as a rotation. This feature therefore allows a string argument to be used as a label, which can be very useful. For example, we can use the * feature to label the following sequence as generating the so-called 'sixspot' configuration (described by Reid):

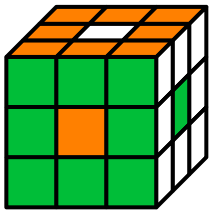
```
\RubikRotation{*sixspot,U,Dp,R,Lp,F,Bp,U,Dp}
```

Alternatively, and probably more conveniently, we could simply use the name 'sixspot' to define a new command, as follows (which therefore allows one to store lots of different rotation sequences by name alone):

```
\newcommand{\sixspot}{U,Dp,R,Lp,F,Bp,U,Dp}
```

With this 'newcommand' we are now able to generate the graphic (sixspot cube) much more easily using the following code:

```
\begin{tikzpicture}[scale=0.7]
   \RubikCubeSolved
   \RubikRotation{\sixspot}
   \DrawRubikCubeRU
\end{tikzpicture}
```



In practice, it is quite useful to go one step further and include the * label feature as well in the newcommand, as follows:

```
\newcommand{\sixspot}{*sixspot,U,Dp,R,Lp,F,Bp,U,Dp}
```

since this has the great advantage of making the particular rotation sequence identified by the label-name visible in the log file. For example, the following command, which uses the rotations **x2** and **y** to initially rotate the 'solved' cube before applying the 'sixspot' sequence of rotations,

```
\RubikRotation{x2,y,\sixspot}
```

will then be represented in the log file as

```
...command=rotation,x2,y,*sixspot,U,Dp,R,Lp,F,Bp,U,Dp
...arguments passed to 'rotation' sub = x2 y *sixspot U Dp R Lp F Bp U Dp (n= 11)
...rotation x OK (= rrR + rrSr + rrLp)
...rotation x OK (= rrR + rrSr + rrLp)
...rotation y OK (= rrU + rrSu + rrDp)
...*sixspot is a label OK
...rotation U OK
...rotation Dp OK
...rotation R OK
...rotation Lp OK
...rotation F OK
...rotation Bp OK
...rotation U OK
...rotation Dp OK
```

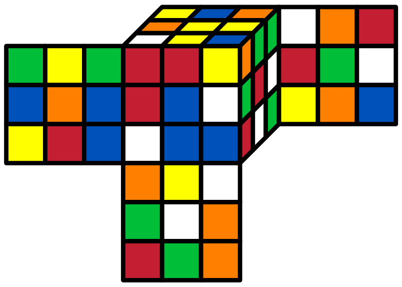
In this way, several named rotation sequences can be easily distinguished in the log file from adjacent rotation commands.

5.1.2 Random rotations

The `\RubikRotation` command can also be used to scramble the cube using a random sequence of rotations. If the first argument is the lowercase word 'random' AND the second argument is an integer n, $(1 \leq n \leq 200)$, then a random sequence of n rotations will be performed; otherwise a default value of 50 is used (for example, if the second argument is not an integer). If $n > 200$ then the value $n = 200$ will be used.

For example, the following commands will scramble a solved cube using a sequence of 120 random rotations, and display the state in the form of a semi-flat cube.

```
\RubikCubeSolved%
\RubikRotation{random,120}%
\begin{tikzpicture}[scale=0.5]
   \DrawRubikCubeFlat
\end{tikzpicture}
```



Note that in this particular example (above), only the `\Draw..` command is inside the TikZ picture environment—a useful method when more than one figure is being drawn. Note also, that when such commands are outside a TikZ picture environment, they should have a trailing % to stop additional white space being included.

The procedure is that all the possible rotations are first allocated a different number (integer) and collected into an array. Then a sequence of n randomised numbers is generated and mapped to the array to generate the associated sequence of random rotations. The sequence used is detailed in the `.log` file.

5.2 `\SaveRubikState`

`\SaveRubikState` The command `\SaveRubikState{`⟨*filename*⟩`}` saves the state (configuration) of the Rubik cube to the file `{`⟨*filename*⟩`}` in the standard `\RubikFace...` format. Consequently such a file can then be input at a later stage so it can be drawn or processed in the usual way (inside the TikZ picture environment).

For example, the following commands would save the so-called 'sixspot' configuration (generated by the rotations **U, Dp, R, Lp, F, Bp, U, Dp**) to the file `sixspot.tex`.

```
\RubikCubeSolved%
\RubikRotation{*sixspot,U,Dp,R,Lp,F,Bp,U,Dp}%
\SaveRubikState{sixspot.tex}%
```

The form of the file `sixspot.tex` will then be as follows—the filename (commented out) is automatically written to the top of the file for convenience.

```
% filename: sixspot.tex
\RubikFaceUp{O}{O}{O}{O}{W}{O}{O}{O}{O}%
\RubikFaceDown{R}{R}{R}{R}{Y}{R}{R}{R}{R}%
\RubikFaceLeft{Y}{Y}{Y}{Y}{B}{Y}{Y}{Y}{Y}%
\RubikFaceRight{W}{W}{W}{W}{G}{W}{W}{W}{W}%
\RubikFaceFront{G}{G}{G}{G}{O}{G}{G}{G}{G}%
\RubikFaceBack{B}{B}{B}{B}{R}{B}{B}{B}{B}%
```

We can therefore access and draw this configuration later, when required, simply by inputting the file as follows:

```
\begin{tikzpicture}
```

```
    \input{sixspot.tex}
    \DrawRubikCubeFlat
\end{tikzpicture}
```

5.3 `\CheckRubikState` command

`\CheckRubikState` Since it is easy to inadvertently define an invalid Rubik cube (eg enter an invalid number of, say, yellow cubies), this command checks the current colour state of all the cubies of a 3x3x3 Rubik cube, and shows the number of cubies of each colour. An ERROR: code is issued if the number of cubies having a given colour exceeds 6. The results are written to the the `.log` file, and displayed under the graphic if the `\ShowRubikErrors` command is used.

One can check the current Rubik state (for errors) by issuing the command

```
\CheckRubikState%
```

Note that such a check is implemented automatically with each `\RubikRotation` command.

5.4 `\ShowRubikErrors` command

`\ShowRubikErrors`

Any errors which arise can be made visible using the command `\ShowRubikErrors`. This command places a copy of the 'error' file (`rubikstateERRORS.dat`) underneath the image so you can see any errors if there are any—all this detail can also be found in the `.log` file.

Consequently, this command must be placed *after* a TikZ picture environment—it cannot be used inside a TikZ environment. In fact this command is probably best placed at the end of the document (if there are several such environments), where it will reveal all rotation errors generated while processing the document.

6 Files generated

Whenever the `\RubikRotation` or `\CheckRubikState` commands are used, three small temporary plain text files for holding data are generated as follows (they are refreshed with each LaTeX run, and are not actively deleted).

- LaTeX writes Rubik state data to the file `rubikstate.dat`.
- Perl reads the file `rubikstate.dat` and then writes the new rubik state to the file `rubikstateNEW.dat`.
- Perl also writes error data to the file `rubikstateERRORS.dat`. A copy of this file is displayed under the image when the command `\ShowRubikErrors` is used after (outside) the TikZ picture environment.

7 General overview

When LaTeX loads the rubikrotation package the following steps are implemented.

1. A check is made to see if `fancyvrb.sty` is loaded: if not then this package is loaded if it is available (this package is required for inputting the file `rubikstateERRORS.dat`).
2. A check is made to see if a configuration file (`rubikrotation.cfg`) exists: if so then this file is input.
3. The text file `rubikstateNEW.dat` is overwritten (if it exists): otherwise the file is created (this prevents an 'old' file being used by LaTeX).
4. The plain text file `rubikstateERRORS.dat` is created. This file collects error messages generated by the Perl script.

When a `\RubikRotation` command is processed it first writes the current colour configuration of each face (the 'rubik state') to the temporary file `rubikstate.dat` (to be read by the Perl script `rubikrotation.pl`). The `\RubikRotation` command also appends the keyword ' `checkrubik`' as well as a copy of the string of Rubik rotations. It then calls the Perl script `rubikrotation.pl`.

For example, if we use the command `\RubikCubeSolved` followed by the command `\RubikRotation{U,D,L,R}` then the associated `rubikstate.dat` file would be as follows:

```
% filename: rubikstate.dat
up,W,W,W,W,W,W,W,W,W
down,Y,Y,Y,Y,Y,Y,Y,Y,Y
left,B,B,B,B,B,B,B,B,B
right,G,G,G,G,G,G,G,G,G
front,O,O,O,O,O,O,O,O,O
back,R,R,R,R,R,R,R,R,R
checkstate
rotation,U,D,L,R
```

Alternatively, if the `\RubikRotation` command was `\RubikRotation{random, 45}` then the last line written to the file would be the string '`rotation,random,45`' A `\CheckRubikState` command triggers the same sequence of events except no 'rotation' line is written.

The action of the Perl program is controlled by the keywords (first argument of each line) associated with each line of the file `rubikstate.dat`. When control passes to Perl, the Perl program starts by loading the rubikstate (prompted by the keywords up, down, left, right, front, back). Next the program performs some basic checks (prompted by the key word 'checkstate'), and then it processes the sequence of Rubik rotations (prompted by the keyword 'rotation'). If, instead, the second argument of the 'rotation' string is the keyword 'random', and provided this is followed by a valid integer, say n, then the Perl program performs a sequence of n random rotations. Finally, the Perl program writes the final 'rubikstate' to the text file `rubikstateNEW.dat`. All error messages are written to the text file `rubikstateERRORS.dat` and also to the LaTeX log file.

Control then reverts to LaTeX which then inputs the file `rubikstateNEW.dat`. If there are more `\RubikRotation` commands then this cycle repeats accordingly. Eventually a '`Draw...`' command of some form is reached and the final rubikstate is drawn in a TikZ picture environment.

If the TikZ picture environment is followed by a `\ShowRubikErrors` command, then a 'verbatim' copy of the `rubikstateERRORS.dat` file is displayed immediately under the graphic. Once the graphic is error-free, then the `\ShowRubikErrors` can be removed or commented out.

8 The code (`rubikrotation.sty`)

8.1 Package heading

```
⟨*rubikrotation⟩
\def\RRfileversion{2.0}%
\def\RRfiledate{2014/01/20}%
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{rubikrotation}[\RRfiledate\space (v\RRfileversion)]
```

The package requires rubikcube.sty. Do not automatically load rubikcube.sty since this makes it difficult to errorcheck new versions (for the moment at least).

```
\@ifpackageloaded{rubikcube}{}{%
   \typeout{---rubikrotation requires the rubikcube package.}%
   }%
```

The rubikrotation package requires access to the fancyvrb package for the `\VerbatimInput{}` command which we use for inputting and displaying the error file.

```
\@ifpackageloaded{fancyvrb}{}{%
   \typeout{---rubikrotation requires the fancyvrb package%
       for VerbatimInput{} command.}%
   \RequirePackage{fancyvrb}}
```

8.2 Some useful commands

`\rubikrotation` First we create a suitable logo

```
\newcommand{\rubikrotation}{\textsf{rubikrotation}}
```

`\@comment` `\@commentone` We require access to the percent character so we can write comments in files, including the log file. We first define percentchar for the write statement (From Abrahams PW, Berry K and Hargreaves KA (1990), "TeX for the Impatient", p 292; available from: CTAN.../info/impatient/)

```
{\catcode`\%=12   \global\def\rubikpercentchar{%}}%
\newcommand{\@comment}{\rubikpercentchar\rubikpercentchar\space}%
\newcommand{\@commentone}{\rubikpercentchar}%
```

`\@print` We need a simple print command for writing comments to a file.

```
\newcommand{\@print}[1]{\immediate\write\outfile{#1}}
```

`\rubikperlname` This holds the name of the Perl-5 script . It is used later to check whether the Perl script exists or not. A plain text configuration file `rubikrotation.cfg` can be used to change the default name of the Perl script using a renewcommand.

```
\newcommand{\rubikperlname}{rubikrotation.pl}
```

`\rubikperlcmd` This holds the commandline code for calling the the Perl script. A plain text configuration file `rubikrotation.cfg` can be used to change the default commandline code using a renewcommand.

```
\newcommand{\rubikperlcmd}{perl \rubikperlname}
```

8.3 Configuration file

If a config file exists (rubikrotation.cfg) then input it here, ie *after* defining the `\rubikperlname` and `\rubikperlcmd` commands and *before* creating the `rubikstateERRORS.dat` file.

```
\typeout{---checking for config file (rubikrotation.cfg)...}
\IfFileExists{rubikrotation.cfg}{%
  \input{rubikrotation.cfg}%
  }{\typeout{---no config file available}%
}%
```

8.4 Clean file rubikstateNEW.dat

We need to clean any existing (old) rubikstateNEW.dat file, since if the TeX shell switch is accidentally not used then Perl will not be CALLed and hence this file will not be renewed (ie an 'old' image may be used).

```
\typeout{---cleaning file rubikstateNEW.dat}%
\newwrite\outfile%
\immediate\openout\outfile=rubikstateNEW.dat%
\@print{\@comment rubikstateNEW.dat (by TeX)}%
\immediate\closeout\outfile%
```

8.5 rubikstateERRORS.dat

The rubikrotation package requires the Perl program. We first open the file `rubikstateERRORS.dat` which is used by the Perl program (`rubikrotation.pl`) for writing its error messages to. This file is accessed and displayed by the command `\ShowRubikErrors`.

IMPORTANT NOTE: this file is created fresh each time LaTeX is run and hence the Perl program only appends data to it during the LaTeX run, so this file just grows until either it is destroyed or recreated—a useful feature to keep since the file accumulates all error messages as the .tex file is processed. We can't make the Perl program create the file since the Perl program is only CALLed if we use a

`\RubikRotation` or `\CheckRubikState` command (which we may not!)—so it has to be created (opened) here.

The following code first opens the file, and then checks to see if the Perl program (`\rubikperlname`) exists; if the Perl prog does exist then all is OK, otherwise we write an error message to the file.

```
\typeout{---creating file rubikstateERRORS.dat}%
\newwrite\outfile%
\immediate\openout\outfile=rubikstateERRORS.dat%
\@print{\@comment rubikstateERRORS.dat}%
\typeout{---checking for Perl script \rubikperlname...}
   \IfFileExists{\rubikperlname}{%
   \typeout{---\rubikperlname\space exists OK}%
   }{\typeout{** ERROR: cannot find Perl program \rubikperlname}%
    \@print{\@comment ** ERROR: cannot find Perl program \rubikperlname}}%
\immediate\closeout\outfile%
```

8.6 Setting up a newwrite and file-access for new files

Having set up all the primary files, we now need to set up a newwrite for all subsequent files openings (eg for rubikstate.dat and saving to arbitrary filenames by the `\SaveRubikState` command). Otherwise, we can easily exceed the LaTeX limit of 15. From here-on TEX will use openout7 when opening and writing to files. We will implement new openings using the command `\@openstatefile` (see below).

```
\typeout{---setting up newwrite for rubikrotation to use...}%
\newwrite\outfile%
```

`\@openstatefile` `\@closestatefile` We also need commands for easy file opening and and closing for new instances of the file `rubikstate.dat` etc. Note that for this we are therefore using the same outfile number as set up by the `\newwrite...` above.

```
\newcommand{\@openstatefile}{\immediate\openout\outfile=rubikstate.dat}
\newcommand{\@closestatefile}{\immediate\closeout\outfile}
```

8.7 Saving the Rubik state

`\@printrubikstate` This command writes the Rubik configuration (state) to the file `rubikstate.dat`, and is used by the `\RubikRotation` command. The file `rubikstate.dat` is read by the Perl program, and represents the state on which the new `\RubikRotation` command acts. Note that we append the key-word `checkstate` to the end of the file in order to trigger the Perl program to implement its `checkstate` subroutine.

```
\newcommand{\@printrubikstate}{%
   \@print{up,\Ult,\Umt,\Urt,\Ulm,\Umm,\Urm,\Ulb,\Umb,\Urb}%
   \@print{down,\Dlt,\Dmt,\Drt,\Dlm,\Dmm,\Drm,\Dlb,\Dmb,\Drb}%
   \@print{left,\Llt,\Lmt,\Lrt,\Llm,\Lmm,\Lrm,\Llb,\Lmb,\Lrb}%
   \@print{right,\Rlt,\Rmt,\Rrt,\Rlm,\Rmm,\Rrm,\Rlb,\Rmb,\Rrb}%
   \@print{front,\Flt,\Fmt,\Frt,\Flm,\Fmm,\Frm,\Flb,\Fmb,\Frb}%
   \@print{back,\Blt,\Bmt,\Brt,\Blm,\Bmm,\Brm,\Blb,\Bmb,\Brb}%
```

```
    \@print{checkstate}%
}
```

8.8 RubikRotation command

\RubikRotation The `\RubikRotation{`⟨*comma separated sequence*⟩`}` command (a) writes the current Rubik state to the file `rubikstate.dat`, and then (b) CALLs the Perl program. It also writes comments to the data file and also to the log file.

```
\newcommand{\RubikRotation}[1]{\IfFileExists{\rubikperlname}{%
    \typeout{---NEW rotation command------------------}%
    \typeout{---command = RubikRotation{#1}}%
    \typeout{---Perl script \rubikperlname\space exists OK}%
    \typeout{---writing current Rubik state to file rubikstate.dat}%
    \@openstatefile% open data file
    \@print{\@comment filename: rubikstate.dat}%
    \@printrubikstate%
    \@print{rotation,#1}%
    \@closestatefile% close data file
    \typeout{---running Perl script}%
    \immediate\write18{\rubikperlcmd}%
    \typeout{---inputting NEW datafile (from Perl)}%
    \input{rubikstateNEW.dat}%
    \typeout{-----------------------------------------}%
 }{\typeout{** ERROR: \rubikperlname\space does not exist}%
}}
```

8.9 ShowRubikErrors command

\ShowRubikErrors This command inputs the file `rubikstateERRORS.dat`.

```
\newcommand{\ShowRubikErrors}{%
    \typeout{---ShowRubikErrors: inputting file rubikstateERRORS.dat}%
    \VerbatimInput{rubikstateERRORS.dat}}
```

8.10 CheckRubikState command

\CheckRubikState This command triggers the Perl program to implement some simple error checking of the Rubik configuration (state). This command (a) writes the current Rubik state to the file `rubikstate.dat`, and then (b) CALLs the Perl program. It also writes comments to the data file and also to the log file..

```
\newcommand{\CheckRubikState}{\IfFileExists{\rubikperlname}{%
    \typeout{---NEW check command------------------}%
    \typeout{---command = CheckRubikState}%
    \typeout{---Perl script \rubikperlname\space exists OK}%
    \typeout{---writing current Rubik state to file rubikstate.dat}%
    \@openstatefile% opens data file
    \@print{\@comment filename: rubikstate.dat}%
    \@printrubikstate%
    \@closestatefile% close data file
    \typeout{---running Perl script}%
    \immediate\write18{\rubikperlcmd}%
    \typeout{---inputting NEW datafile (from Perl)}%
    \input{rubikstateNEW.dat}%
    \typeout{-----------------------------------------}%
  }{\typeout{** ERROR: \rubikperlname\space does not exist}%
}}
```

8.11 SaveRubikState macro

`\SaveRubikState` The command `\SaveRubikState{`⟨*filename*⟩`}` saves the Rubik state to a file. Note that in order to actually write a LaTeX command to a file without a trailing space one must use the `\string` command (see the book "TeX by Topic", p 238). Note that this macro uses the two internal commands `\Rubik@comment` and `\Rubik@print`. #1 is the output filename. We use several `\typeout` commands to write to the log file.

```
\newcommand{\SaveRubikState}[1]{%
\typeout{---NEW save command------------------}%
\typeout{---command = SaveRubikState{#1}}%
\typeout{---saving Rubik state data to file #1}%
\immediate\openout\outfile=#1%
\@print{\@comment filename: #1\@commentone}%
\@print{\string\RubikFaceUp%
    {\Ult}{\Umt}{\Urt}{\Ulm}{\Umm}{\Urm}{\Ulb}{\Umb}{\Urb}\@commentone}%
\@print{\string\RubikFaceDown%
    {\Dlt}{\Dmt}{\Drt}{\Dlm}{\Dmm}{\Drm}{\Dlb}{\Dmb}{\Drb}\@commentone}%
\@print{\string\RubikFaceLeft%
    {\Llt}{\Lmt}{\Lrt}{\Llm}{\Lmm}{\Lrm}{\Llb}{\Lmb}{\Lrb}\@commentone}%
\@print{\string\RubikFaceRight%
    {\Rlt}{\Rmt}{\Rrt}{\Rlm}{\Rmm}{\Rrm}{\Rlb}{\Rmb}{\Rrb}\@commentone}%
\@print{\string\RubikFaceFront%
    {\Flt}{\Fmt}{\Frt}{\Flm}{\Fmm}{\Frm}{\Flb}{\Fmb}{\Frb}\@commentone}%
\@print{\string\RubikFaceBack%
    {\Blt}{\Bmt}{\Brt}{\Blm}{\Bmm}{\Brm}{\Blb}{\Bmb}{\Brb}\@commentone}%
\immediate\closeout\outfile%
\typeout{-----------------------------------------}%
}%
```

——————— End of this package ———————

```
⟨/rubikrotation⟩
```