projector: a simple LaTeX document class for making slides

Victor Shoup

January 15, 2009

1 Introduction

There are other LaTex classes that do very similar things, such as beamer and prosper, so why another class? I wrote this class because I found it easier and more interesting than reading all the documentation and/or customizing these other classes.

projector allows one to make presentations that can be viewed using any standard PDF viewer. It allows "incremental builds" of slides, as a sequence of "overlays". While it is easy to customize the color scheme, layout, etc., the package is designed to produce rather simple looking slides, without a lot of gratuitous splashes, streaks, or other decorations, and without fancy transitions that sparkle or bounce.

projector is intended to be simple and extensible. It provides basic functionality, and does not impose any preconceived notions about what your slides should look like.

As for the name: "projector" is the English word for the German word "beamer".

2 Installing

These instructions assume you want to install into your "personal texmf tree", rather than into a "system texmf tree".

Before doing anything, you have to determine where your "personal texmf tree" is. On Linux, this is usually ~/texmf. On Mac OSX, this is usually ~/Library/texmf. In what follows below, I'll write $TEXMF for your "texmf tree".

To install the projector class, simply copy the file projector.cls to $TEXMF/tex/latex. For some (but not all) latex distributions, you may also have to run the command texhash afterwards.

You may want to use some packages that provide special font support. To find out if package foo is already installed, run the command

```
kpsewhich foo.sty
```

If this prints out a path name, you already have it. If not, the package should be in the accompanying `support` directory, and installation directions are given below.

One package that you may want to install is `sfmath`. If you don't have it, just copy the file `sfmath.sty` from `support` to `$TEXMF/tex/latex` (and follow with `texhash`). This package allows you to typeset all LaTex math in sans serif font.

Another package called `arev` provides a sans serif font for all text (normal and math) that is particularly nice to read in presentations. Unfortunately, it takes just a bit more work to install this package, if you don't already have it. Here are the instructions.

1. `cd` to `$TEXMF/..` and copy `arev-texmf.tgz` from `support` to here
2. run `tar -xzf arev-texmf.tgz`

   If your `tar` is not `gnutar`, you'll need to replace the above `tar` command by `gunzip -c arev-texmf.tgz | tar xf -`
3. Finally, you'll need some extra magic to complete the font installation. What seems to work is the following:

```
updmap --enable Map arev.map
updmap --enable Map bera.map
updmap --enable Map mdbch.map
```

How these fonts may be used in your LaTex documents is discussed below.

3 Getting started

3.1 The basic LaTex document structure

```
\documentclass{projector}

  % preamble -- load packages and define macros here

\begin{document}

\begin{slide}
  ...
\end{slide}
     .
     .
     .
\begin{slide}
  ...
\end{slide}

\end{document}
```

3.2 Processing and viewing

- Process your LaTex file using `pdflatex`, which produces a PDF file.
- View your PDF with any standard PDF viewer, and use "full screen mode" when you want to view it using a projector.

3.3 Fonts

A good choice of fonts is essential for a good presentation. Unfortunately, the default CM (Computer Modern) fonts are *not* very appropriate. Usually, a sans serif font is preferable.

One simple hack that requires no special font packages is to place the following in the preamble:

```
\renewcommand\rmdefault{\sfdefault}
```

This will typeset all normal text as CM sans serif, but all math text will still be typeset in normal CM math font. Some people like the look of this, but to make both normal and math text typeset in CM sans serif, place the following in the preamble:

```
\usepackage{sfmath}
\renewcommand{\rmdefault}{\sfdefault}
```

The `sfmath` package is easily installed, as described above, if you don't already have it. For a somewhat heavier look, try the following:

```
\usepackage{helvet}
\usepackage{sfmath}
\renewcommand{\rmdefault}{\sfdefault}
```

My preferred method, though, is the following:

```
 \usepackage{arev}
```

If you don't already have package `arev` installed, it should not be too difficult to do so, as described above. The `arev` font was specifically designed to look very nice in presentations. It is a sans serif font with a nice, heavy look that is easy to read when projected.

3.4 The basic slide environment

```
\begin{slide}

Body.

\end{slide}
```

Note that the text in a slide is not automatically centered, either vertically or horizontally. For example, to make the above text centered both vertically and horizontally, write:

```
\begin{slide}

\vfill

\begin{center}
Body.
\end{center}

\vfill

\end{slide}
```

Also, note that text will be formatted with a "ragged right edge", and with no paragraph indentation. As such, a small amount of space is automatically inserted between paragraphs. If you want a line break without the extra paragraph spacing, use \\.

There is no automatic mechanism for generating slide titles (but see §5.10 for some helpful title-making commands).

There is also a `slide*` environment, which is useful when "verbatim text" needs to be included on a slide. It is also useful for debugging. See §5.17 below.

3.5 Basic overlay usage

```
\begin{slide}
First item

\pause
Second item

\pause
Third item
\end{slide}
```

This will produce a slide that expands into three overlays (i.e., 3 PDF pages). The three lines of text will be revealed one at a time. You don't have to explicitly specify the number of overlays: this is automagically deduced.

Here is an example of a bulleted list, revealed a bit at a time:

```
\begin{slide}
\begin{itemize}
\pause\item Line 1
\pause\item Line 2
\pause\item Line 3
\pause\item Line 4
\end{itemize}
\end{slide}
```

For convenience, there is a command `\pitem` that is equivalent to `\pause\item`, so the above can be written:

```
\begin{slide}
\begin{itemize}
\pitem Line 1
\pitem Line 2
\pitem Line 3
\pitem Line 4
\end{itemize}
\end{slide}
```

The overlay mechanism is implemented in a simple way using colors. Indeed, each instance of `\pause` (and all of the other overlay commands discussed below) quite literally translates into a command `\color{c}`, where `c` is determined by context. In particular, just like the usual `\color` command, the effect of these commands extends to the end of the smallest enclosing scope.

Behind the scenes, here is what happens. Suppose the slide expands into N overlays. Then the LaTex code on the slide is processed N times. There are two counters: `overlaynum` and `pausecnt`. The counter `overlaynum` is initialized to 1, and incremented as each overlay is processed. The counter `pausecnt` is initialized 1 when starting to process each overlay. The semantics of `\pause` is:

```
pausecnt++
if overlaynum >= pausecnt then \visible else \invisible
```

Here, `\visible` is a macro that expands into a `\color` command that makes the following text visible, while `\invisible` expands into a `\color` command that makes the following text blend into the background. The effect of this `\color` command extends from here to the end of scope.

These rules are at least fairly simple and clear, and usually pretty convenient. However, sometimes the "scoping effect" can be inconvenient. To extend the effect of a `\pause` command beyond its scope, you can use the `\refresh` command. For example, in a `tabular` environment, to reveal one row at a time, you should write:

```
\begin{tabular}{ccc}
\pause A & \refresh B & \refresh C \\
\pause D & \refresh E & \refresh F
\end{tabular}
```

Without the \refresh's, B and C would still be visible, because the scope of the first \pause is limited to A (this is an artifact of the implementation of tabular). In fact, you can avoid writing so many \refresh's using a feature of the included colortbl package (which is loaded by the projector class).

```
\begin{tabular}{>{\refresh}c>{\refresh}c>{\refresh}c}
\pause A &  B & C \\
\pause D & E & F
\end{tabular}
```

Or even shorter, using the usual repeat mechanism of tabular:

```
\begin{tabular}{*{3}{>{\refresh}c}
\pause A &  B & C \\
\pause D & E & F
\end{tabular}
```

Here is another example of combining \pause and \refresh commands in an align* environment, from the amsmath package (also loaded by projector):

```
\begin{align*}
\pause X & \refresh = Y + Z \\
\pause W & \refresh \le A + B \\
  & \pause = A' + B' \\
  & \pause \le A'' + ''B
\end{align*}
```

The \pause command also takes an optional argument that specifies the number overlays you wish to pause (default = 1).

3.6 Using colors

Because the overlay mechanism is implemented using colors, one has to be a bit careful in how one uses colors in a document.

Use the command \fgcolor{#1}, either between slides or within a slide, to set the foreground color to #1, e.g., \fgcolor{red}. The effect of this command extends to the end of the enclosing scope. You may reference the current foreground color using the special color name foreground.

One should in general avoid direct use of the \color command, using \fgcolor in its place.

Use the command \bgcolor{#1}, either between slides or within a slide, to set the background color to #1. If used between slides, all subsequent slides are affected; if used within a slide, only that slide is affected. You may reference the current background color using the special color name background.

The commands \colorbox and \fcolorbox should be avoided. Instead, use the commands \Colorbox and \FColorbox, which have the same syntax, but which are designed to work with

the overlay mechanism: if the entire box is currently invisible, then it will disappear into the background; moreover, the overlay mechanism works within such a box as well; in particular, `\pause` commands within a box work as expected.

There is also an environment `BOX`, described below in §5.11, which generalizes and extends the functionality of `\Colorbox` and `\FColorbox`.

Use the command `\filtered{#1}` in any other context where you want a color `#1` to be properly filtered using the overlay mechanism.

Note that the package `xcolors` is loaded by `projector`, and so you can use write "color mixes", like `red!50!yellow`, in place of color names, with one exception: the argument to the `\filtered` command should always be a simple color name (use the `\colorlet` to define a new color name, if necessary). In addition, the `xcolor` package is loaded with both the `x11names` and `svgnames` options, which predefines a whole host of useful color names. See the manual for the `xcolor` package for more details.

4 More advanced overlays

4.1 The `\hideX` commands

There are commands `\hideTo`, `\hideFrom`, and `\hideAt`. Each takes an overlay number as an argument. For example, the command `\hideAt{2}` will make things invisible at overlay 2, and otherwise does nothing.

More precisely, the semantics of `\hideTo{#1}` (resp., `\hideFrom{#1}`, `\hideAt{#1}`) are the following: if the relevant inequality is satisfied (i.e., `overlaynum` $\leq$ `#1` (resp., $\geq$ `#1`, $=$ `#1`)), then the `\invisible` command is executed, and otherwise, the command has no effect at all. The effect, if any, extends to the end of the enclosing scope.

4.2 The `\showX` commands

Similarly, there are commands `\showTo`, `\showFrom`, and `\showAt`. So for example, `\showAt{2}` will make things visible at overlay 2, but has no effect otherwise.

The semantics are that if the relevant inequality is satisfied, the `\visible` command is executed, and otherwise, the command has no effect at all. The effect, if any, extends to the end of the enclosing scope.

For example, to make some text appear only on overlays 3 through 6

```
\invisible\showFrom{3}\hideFrom{7} ... some text ...
```

or alternatively:

```
\invisible\showTo{6}\hideTo{2} ... some text ...
```

In processing such a sequence of `\showX`/`\hideX` commands, it is the last inequality that is satisfied that determines the result.

4.3 The \ShowX and \HideX commands

For convenience, there are also commands \ShowTo, \ShowFrom, and \ShowAt. The command \ShowX{#1} is equivalent to \invisible\showX{#1}. The semantics are that if the relevant inequality is satisfied, the \visible command is executed, and otherwise, the \invisible command is executed. Thus, the command *always* has an effect, which extends to the end of the enclosing scope.

Example:

```
\begin{slide}
{\ShowAt{3} A line visible only on overlay 3}

{\ShowFrom{2} A line visible only on overlays 2 and 3}

{\ShowTo{2} A line visible only on overlays 1, 2}
\end{slide}
```

Similarly, there are commands \HideTo, \HideFrom, and \HideAt. The command \HideX{#1} is equivalent to \visible\hideX{#1}. The semantics are that if the relevant inequality is satisfied, the \invisible command is executed, and otherwise, the \visible command is executed. Thus, the command *always* has an effect, which extends to the end of the enclosing scope.

The \pause command is in fact simply implemented as \PauseStep\ShowFrom{\p}. Here, the \PauseStep command increments the pause counter. It takes an optional argument (default = 1), indicating how much to increment the counter. There is also a command \PauseSet: executing \PauseSet{#1} sets the pause counter to #1. It is best to modify the pause counter only through these two commands.

4.4 The \colorX commands

There are commands \colorTo, \colorFrom, and \colorAt. For example, the command

```
\colorAt{2}{red}
```

makes things red at overlay 2, and otherwise does nothing. Again, the effect extends to the end of the enclosing scope.

4.5 The \alertX and \alert commands

There are commands \alertTo, \alertFrom, and \alertAt. For example, the command

```
\alertAt{2}
```

will make the foreground color a special "alert color" at overlay 2.

The command `\alert` unconditionally makes the foreground color equal to the "alert color". Again, the effect extends to the end of the enclosing scope.

You may change the alert color by setting the style parameter `alert.color`. For example:

```
\style{alert.color=green}
```

sets the alert color to green. `alert.color` is one of many "style parameters" that can by tuned. For a complete discussion, see §6.6 below.

4.6 The `\fadeX` and `\fade` commands

There are commands `\fadeTo`, `\fadeFrom`, and `\fadeAt`. For example, the command

```
\fadeAt{2}{20}
```

makes things appear 20% faded (instead of he default 0%) at overlay 2, and otherwise does nothing.

The command `\fade{20}` fades things out unconditionally. Again, the effect extends to the end of the enclosing scope.

The macro `\fadeamount` represents the current "fade amount". This is useful for performing relative fades. For example, to decrease the *contrast* by 25%, you can write:

```
\FPeval\newfade{clip(100 - 0.75*(100 - fadeamount))}
\fade{\newfade}
```

This makes use of the `fp` package, which is loaded by `projector`.

4.7 Overlay numbers

The argument to `\hideX`, `\showX`, `\HideX`, `\ShowX`, `\colorX`, `\alertX`, `\fadeX` need not be a constant. In fact, it may be a simple expression, as recognized by the `calc` package (which is loaded by `projector`). Moreover, within this expression, one may write `\p` to refer to the current value of `pausectr`. (Note that the macro `\p` is *locally* defined as `\value{pausectr}`, and won't interfere with any other definitions of `\p`). This can be conveniently used for many special effects. For example:

```
\newcommand{\rollitem}{\pause\hideFrom{\p+2}\item}

   ...

\begin{itemize}
\rollitem a
\rollitem b
\rollitem c
\end{itemize}
```

This creates a bulleted list where each item is displayed for two overlays, creating a "rolling" effect.

As another example, consider:

```
\newcommand{\alertiteminit}{\colorlet{savedfg}{foreground}}
\newcommand{\alertitem}{\fgcolor{savedfg}\pause\alertAt{\p}\item}

   ...

\begin{itemize} \alertiteminit
\alertitem a
\alertitem b
\alertitem c
\end{itemize}
```

This creates a bulleted list where each item is revealed one at a time, and is also highlighted in the "alert color" when it is revealed.

One more list example:

```
\newcommand{\fpitem}{\pause\fade{0}\fadeFrom{\p+1}{50}\item}

   ...

\begin{itemize}
\fpitem A
\fpitem B
\fpitem C
\end{itemize}
```

This creates a bulleted list where each item appears one at a time, and fades out when the next item appears.

As another example, consider:

```
${\hideAt{\p}%
    \underbrace{%
       {\showAt{\p}v_1, v_2, \ldots, v_{k-2}}%
     }_{\text{moved nodes}}}, v_{k-1}, v_k$
\pause
```

This has the effect of showing just $v_1, \ldots, v_{k-2}, v_{k-1}, v_k$ on one overlay, and then additionally shows the underbrace and the subscripted text on the next overlay.

Also, the number of overlays for a given slide is calculated as the maximum overlay number appearing as an argument to any command on that slide.

Usually, you will get just the right number of overlays. If you want to force at least n overlays, just place the command \showFrom{n} at the end of the slide. If for some reason you have more overlays than you want, place the command \trimslide at the end of the slide. \trimslide takes an optional argument that specifies how many overlays you want to trim (default=1). Supplying a negative argument works as you would expect, adding overlays, instead of trimming them.

4.8 \ifvisible and visibleP

The command \ifvisible{<then part>}{<else part>} executes the <then part> if the text is currently visible, and executes the <else part> otherwise. The corresponding value \boolean{visibleP} may be used in \ifthenelse tests (from the ifthen package, which is loaded by projector). Note that this also defines a low-level tex "conditional" \ifvisibleP.

4.9 The \refresh and \record commands

The \refresh command actually works as follows: it restores the "local visibility status" that was in effect just after the last \visible or \invisible command on the current overlay was executed (with an implicit \visible at the beginning of each overlay).

There is even a more general mechanism for working around "scoping" issues. The command \record{TAG} records the current "local visibility status". The tag name TAG may be supplied as an optional argument to \refresh, so that \refresh[TAG] restores the "local visibility status" that was in effect when \record{TAG} was executed. The \visible and \invisible commands actually work by recording to the special tag name @, and \refresh is equivalent to \refresh[@].

4.10 Making hidden text partially visible

By default, "invisible" items are the same color as the background. You can make these items "partially visible" by setting the style parameter hidden. For example:

```
\style{hidden=90}
```

This makes the hidden text only "90% hidden". This is sometimes preferable to the default, as it gives viewers the chance to read ahead, if they want.

Combining primitives, one can easily build some nice special effects. For example, consider the following. Let \alertitem and \alertiteminit be defined as above (in §4.7). Consider:

```
\begin{itemize} \alertiteminit \style{hidden=0}
\alertitem a
\alertitem b
\alertitem c
\end{itemize}
```

This will highlight each item in the "alert color" one at a time, while all items remain visible.

4.11 Turning overlays off

The command

```
\overlaysoff
```

turns off the overlay mechanism. When this is done, the commands

```
\pause,      \PauseStep,    \PauseSet
\visible,    \invisible
\hideTo,     \hideFrom,     \hideAt,
\showTo,     \showFrom,     \showAt,
\HideTo,     \HideFrom,     \HideAt,
\ShowTo,     \ShowFrom,     \ShowAt,
\colorTo,    \colorFrom,    \colorAt,
\alertTo,    \alertFrom,    \alertAt
\fadeTo,     \fadeFrom,     \fadeAt
\refresh,    \record
\trimslide
```

have no effect at all. The impact of `\overlaysoff` extends to the end of scope. There is no `\overlayson` command.

Typically, you include this command in the preamble when you want to prepare a version for printing. The result will be that each slide consists of a single overlay, with all text on the slide visible.

There is also a command

```
\ifoverlaysoff{<then part>}{<else part>}
```

which will expand the `<then part>` when the overlay mechanism is turned off, and the `<else part>` otherwise. This is useful for certain special effects. The corresponding boolean value `\boolean{overlaysoffP}` may also be used in `\ifthenelse` tests. Note that this also defines a low-level tex "conditional" `\ifoverlaysoffP`.

5 Carrying on

5.1 "Hot corners" for navigation

When viewing using a standard PDF viewer, the last overlay of each slide has a "hypertext anchor". Clicking on the lower left corner of any page will bring you to the previous anchor, while clicking on the lower right corner will bring you to the next anchor. This allows you to quickly skip over intermediate overlays of a slide. Note that there is no visible marker on the slide for where to click: you just have to point the mouse in the corner, and click.

5.2 Hyperlinks

The class `projector` loads the package `hyperref`, which defines the command `\hyperlink`. The command

```
\hyperlink{TARGET}{some text}
```

will turn some "some text" into a hyperlink: when you click on it, your browser should jump to the page with a target called `TARGET`. To set the target slide, use the command `\targetslide`:

```
\begin{slide}
\targetslide{TARGET}
...
\end{slide}
```

By default, `\targetslide` makes the last overlay of a slide the actual target. However, `\targetslide` takes an optional parameter which specifies the overlay of the slide is the actual target. So `\targetslide[2]{TARGET}` makes the second overlay the target. The overlay number may be an arithmetic expression that includes `\p`, which represents the current value of the pause counter.

Also, the hyperref package defines the `\url` command so that it generates an external hyperlink.

5.3 Stretching and shrinking

You can stretch/shrink certain vertical spacing parameters (space between items in a list, paragraphs, and also `\smallskip`, `\medskip`, and `\bigskip`) by setting the style parameter `vskip`. For example:

```
\style{vskip=75}
```

sets the vertical spacing to 75% of the default, allowing more text to fit on a slide.

Additionally, there is a command

```
\squeeze
```

which will make all of these vertical skips "shrinkable", allowing more material to be squeezed onto a slide. This command obeys usual scoping rules. If you are lazy, just put `\squeeze` in the preamble, so it stays in effect for the whole file.

NOTE: if you want to squeeze a single slide, or change the value of the style parameter `vskip` for a single slide, for annoying TeXnical reasons, you are better off doing this as

```
{\squeeze \style{vskip=...}
\begin{slide}
...
\end{slide}
}
```

rather than

```
\begin{slide}
\squeeze \style{vskip=...}
...
\end{slide}
```

5.4 Page layout

The margins of a slide can be adjusted via the style parameter `slide.margin`. For example, to make the all margins 200% of the default, do:

```
\style{slide.margin=200}
```

You can similarly adjust the top, bottom, and side margins individually by setting the style parameters `slide.margin.top`, `slide.margin.bot`, and `slide.margin.side`.

The style parameter `slide.size` controls the amount of text that fits on a slide. For example:

```
\style{slide.size=120}
```

makes the slide size 120% of the default, which has the effect of making the text appear smaller when projected.

The style parameter `slide.aspect.ratio` controls the aspect ratio of slides. The default is `4:3`. For example:

```
\style{slide.aspect.ratio=16:9}
```

gives a "wide screen" aspect ratio.

Unlike all other style parameters, all of these `slide....` style parameters should be set only in the preamble to have the desired effect.

5.5 Dimensions

As usual, `\textheight` and `\textwidth` are lengths representing the height and width of the text, not including the margins, and `\paperheight` and `\paperwidth` represent the height and width of the entire page, including margins. In addition, `\smargin`, `\tmargin`, and `\bmargin` are the lengths of the slide margins (side, top, and bottom). Also, `\fsmargin`, `\ftmargin`, and `\fbmargin` are the lengths of the footer margins, and `\footwidth` is the width of the footers (see below). None of these dimensions should be directly modified by the user; rather, appropriate style parameters should be set, which automatically redefines these dimensions.

5.6 Footers and Headers

You can add running footers and headers that appear in the margins at the top and bottom of each slide.

To create customized footers and headers, set the style parameters

```
foot.text.top.left   foot.text.top.center  foot.text.top.right
foot.text.bot.left   foot.text.bot.center  foot.text.bot.right
```

The counter `slidenum` represents the current slide number. So to make a footer with a slide number in the lower right corner, do the following:

```
\style{foot.text.bot.right=\theslidenum}
```

The macro `\lastslide` represents the total number of slides in your document, so you can write

```
\style{foot.text.bot.right=\theslidenum/\lastslide}
```

to give the audience a warning of how many slides are left.

You can use the command `\fgcolor` within such customized text, but overlay commands (`\pause`, `\showX`, `\colorX`, ...) have no effect.

By default, all footers and headers are formatted using the current foreground color and "tiny" font. To change the font or color of all footers and headers, set the style parameter `foot.format`. By default, this is empty.

For example, suppose you want just lightly visible footers:

```
\style{foot.format={\fgcolor{background!90!foreground}}}
```

Note the use of the color mix `background!90!foreground`, which creates a blend of 90% of the current background color and 10% of the current foreground color.

The margins of footers and headers on the slide are controlled via the style parameter `foot.margin`. For example, to make the all footer and header margins 200% of the default, do:

```
\style{foot.margin=200}
```

You can similarly adjust the top, bottom, and side footer margins individually by setting the style parameters `foot.margin.top`, `foot.margin.bot`, and `foot.margin.side` (footers on the top are usually called "headers").

The footer margin measures the distance from the edge of the page to the footer text. Note that slide and footer margins are completely independent. If the slide margins are too small, the footer text will protrude into the body of the slide.

There is no built-in mechanism for adding ruled lines to headers and footers, but this is easily accomplished:

```
\style{foot.text.top.left=
   {\makebox[0pt][l]{\raisebox{-2pt}{\downbox{\rule{\footwidth}{0.5pt}}}}%
   ...text... }
}
\style{foot.text.bot.left=
   {\makebox[0pt][l]{\raisebox{\heightof{X}+2pt}{\rule{\footwidth}{0.5pt}}}%
   ...text... }
}
```

The above generates ruled lines that extend the width of the headers and footers. In addition to standard LaTeX commands, it makes use of the command `\downbox`, discussed below (see §5.7). To make lines that extend across the entire page, do the following:

```
\style{foot.text.top.left=
   {\makebox[0pt][l]{\raisebox{-2pt}{%
      \downbox{\hspace{-\fsmargin}\rule{\paperwidth}{0.5pt}}}}%
   ...text... }
}
\style{foot.text.bot.left=
   {\makebox[0pt][l]{\raisebox{%
      \heightof{X}+2pt}{\hspace{-\fsmargin}\rule{\paperwidth}{0.5pt}}}%
   ...text... }
}
```

The interface for setting margins in terms of percentages of their initial defaults is intended to be convenient. However, in some situations, it can be slightly awkward. Suppose, for example, we want to make the side footnote margins equal to the slide side margin. One can still do this, as follows:

```
\dimdiv{\mratio}{\smargin}{\fsmargin}
\FPmul{\mratio}{\mratio}{100}
\style{foot.margin.side=\mratio}
```

Here, `\dimdiv{#1}{#2}{#3}` is a command provided by the `projector` class, which computes `#1 := #2 / #3`, where `#2` and `#3` are dimensions, and `#1` is a decimal number. `\FPmul{#1}{#2}{#3}` is a command provided by the `fp` package (loaded by the `projector` class), which computes `#1 := #2 * #3`, where all parameters are decimal numbers.

5.7 Vertical box placement

When making slides, one often wants to string boxes of graphics and text together on a line. For convenience, commands `\upbox`, `\downbox`, and `\centerbox` are provided. Each of these commands typesets its argument as a box, and then shifts the baseline of the box so that it is at the bottom (for `\upbox`), at the top (for `\downbox`), or centered (for `\centerbox`). When such boxes are placed on a line, their new baselines line up the baseline of the current line of text.

For example,

```
\downbox{\includegraphics{foo1}} \downbox{\includegraphics{foo2}}
```

will produce a line of text with the top edge of each graphic box aligned at the baseline of the current line of text.

```
\centerbox{\includegraphics{foo1}} \centerbox{\includegraphics{foo2}}
```

will align the center of each graphic box with the baseline of the current line of text.

Each of these commands takes an optional length argument, which shifts the box up by the given amount, after performing the initial shift. For example,

```
\centerbox[0.5ex]{\includegraphics{foo1}} implies
\centerbox[0.5ex]{\includegraphics{foo2}}
```

will center the graphic box around a point `0.5ex` above the baseline of the current line of text, which may be more visually appealing in some cases.

These commands are quite simply implemented in terms of the `\raisebox` command, as follows:

```
\newcommand{\upbox}[2][0pt]{\raisebox{\depth+(#1)}{#2}}
\newcommand{\downbox}[2][0pt]{\raisebox{-\height+(#1)}{#2}}
\newcommand{\centerbox}[2][0pt]%
   {\raisebox{(\depth-\height)*\real{0.5}+(#1)}{#2}}
```

5.8 Arbitrary box placement

Suppose you just want to place one or more objects at arbitrary places on your slide. Use the command `\putbox` for this. This command is invoked as

```
\putbox{<hoffset>}{<voffset>}{<text>}
```

`<text>` is placed on the slide at a horizontal distance of `<hoffset>` and a vertical distance of `<voffset>` from the lower left-hand corner of the slide. Thus, `\putbox{0pt}{0pt}{X}` puts an `X` is the lower left-hand corner, while `\putbox{0.5\textwidth}{0.5\textheight}{X}` puts an `X` near the middle. In the latter example, to completely center the `X`, both horizontally and vertically, write

```
\putbox{0.5\textwidth}{0.5\textheight}{\makebox[0pt][c]{\centerbox{X}}}
```

Here are some other recipes:

```
\putbox{\textwidth}{0pt}{\makebox[0pt][r]{X}} % lower right
\putbox{\textwidth}{\textheight}{\downbox{\makebox[0pt][r]{X}}}
  % upper right
\putbox{\textwidth}{0.5\textheight}{\centerbox{\makebox[0pt][r]{X}}}
  % centered right
```

Note that <text> may contain arbitrary commands, including overlay commands like \ShowX, \HideX, etc. These commands are processed when the \putbox command is processed. You may place as many \putbox commands as you like in a slide. They do not affect in anyway the placement of ordinary text on the slide. The boxes generated will appear on top of ordinary text.

5.9 Conditional typesetting

Sometimes, you may want to show one thing on the first overlay of a slide, and another thing on the second. However, you want to make sure both things take up the same amount of space; otherwise, the surrounding text will move around.

The \ifthenbox command takes care of the details. When you write

```
\ifthenbox{<test>}{<then part>}{<else part>}
```

both the <then part> and <else part> are evaluated, and a box is created that is just large enough to hold both of them. The <test> determines which part goes in the box.

The <test> may be any test recognized by the usual \ifthenelse command from the ifthen package. Also, the macro \n is locally defined as the current overlay number (\n = \value{overlaynum}), and the macro \p is locally defined as the current value of the pause counter (\p = \value{pausectr}).

\ifthenbox commands may be arbitrarily nested.

Here is an example:

```
Hello
\ifthenbox{\NOT\n>\p}{%
   $x + y + x + z$%
}{%
\ifthenbox{\n=\numexpr\p+1}{%
   \parbox[b]{50pt}{HELLO OUT THERE}%
}{%
   $\displaystyle \sum_{i=1}^n i^2$
}}
World

\pause[2]
```

Up through overlay number \p, the first expression is displayed. At overlay number \p+1, the second expression is displayed. After overlay number \p+1, the third is displayed (note the use of \numexpr — this is necessary for nontrivial arithmetic expressions within an \ifthenelse). Enough space is allocated for all three expressions, and the baselines will line up nicely.

A common usage is conditional inclusion of graphics files (i.e., PDF files). Since one cannot easily make such graphics disappear by changing their color, one simply has to not include them to make them invisible. However, one has to allocate sufficient space even when the graphics is not included. This is trivial to do using \ifthenbox. For example:

```
\ifthenbox{\boolean{visibleP}}{\includegraphics[scale=0.5]{foo}}{}
```

This is such a common case that a special command `\graphicbox` is included, which is simply defined as follows:

```
\newcommand{\graphicbox}[2][]{%
    \ifthenbox{\boolean{visibleP}}{\includegraphics[#1]{#2}}{}%
 }
```

Thus, we could write the previous example as follows:

```
\graphicbox[scale=0.5]{foo}
```

If we want to center the box on the current line, we could write:

```
\centerbox[0.5ex]{\graphicbox[scale=0.5]{img}}
```

The following example shows how to make a slide consisting of two overlays, with one graphic for the first, another for the second, and with only the second appearing in the print version.

```
\begin{slide}
\vfill
\begin{center}
\ifthenbox{\boolean{overlaysoffP} \OR \n=2}{%
  \includegraphics{bar}%
}{%
  \includegraphics{foo}%
}
\end{center}
\pause
\vfill
\end{slide}
```

Note that the `\ifthenbox` takes an optional position parameter, which is `l`, `c`, or `r`. This controls the horizontal positioning of individual boxes within the resulting, larger box. The default is `c` (for centered). If you want all boxes left justified, write `\ifthenbox[l]{...}{...}`. If you want all boxes right justified, write `\ifthenbox[r]{...}{...}`.

5.10 Banners and stripes

The commands

```
\banner{<text>}, \Banner{<text>}, \BANNER{<text>}
```

are useful for providing slide titles (of increasing levels of emphasis).

Banner colors and fonts can be changed by setting the style parameters:

```
banner.format, Banner.format, BANNER.format
banner.env, Banner.env, BANNER.env
banner.aboveskip, Banner.aboveskip, BANNER.aboveskip
banner.belowskip, Banner.belowskip, BANNER.belowskip
```

The command

```
\Stripe{<text>}
```

makes a title in a colored stripe across the top of the slide. This command should be issued only at the beginning of a slide environment.

Stripe colors, fonts, and layout can be changed by setting the style parameters:

```
Stripe.margin, Stripe.bgcolor, Stripe.env, Stripe.format
```

5.11 Fancy Colored Boxes

The BOX environment is a convenient way of typesetting text in a box that may have a colored background and/or frame and/or a drop shadow.

In its basic form, you write

```
\begin{BOX}
some text
\end{BOX}
```

The behavior of BOX is governed by several style parameters:

```
parameter      default           meaning

BOX.fwidth     100               frame width (% of default)
BOX.margin     100               margin (% of default)
BOX.bgcolor    blue!20           background color
BOX.fcolor     foreground        frame color
BOX.framed     false             box has a frame
BOX.opaque     false             box background is colored
BOX.shaded     false             box has a drop shadow
BOX.extended   false             margins extend into surrounding text
BOX.when       \boolean{true}    condition to trigger frame/background/shadow
```

One can set these style parameters as usual, or one can override the defaults locally for a single BOX. For example, to make a yellow box with a blue frame and a drop shadow, you could write:

```
\style{BOX.bgcolor=yellow, BOX.opaque=true, BOX.fcolor=blue,
       BOX.framed=true, BOX.shaded=true }

\begin{BOX}
some text
\end{BOX}
```

Or you could write this as:

```
\begin{BOX}[bgcolor=yellow, opaque=true, fcolor=blue, framed=true, shaded=true]
some text
\end{BOX}
```

Or more simply as:

```
\begin{BOX}[bgcolor=yellow, opaque, fcolor=blue, framed, shaded]
some text
\end{BOX}
```

Or even more simply as:

```
\begin{BOX}[bgcolor=yellow, fcolor=blue, shaded]
some text
\end{BOX}
```

The rule is: locally setting `bgcolor` sets `opaque=true`, and locally setting either `fcolor` or `fwidth` sets `framed=true`.

The overlay mechanism will work correctly both outside and inside a `BOX` environment.

The `when` parameter is a test that determines if the frame, background color, and drop shadow are actually are shown. This can be useful for highlighting text at appropriate times. For example, to highlight at overlay 5:

```
\begin{BOX}[bgcolor=yellow, fcolor=blue, shaded, when={\n=5}]
some text
\end{BOX}
```

The value of `when` is a boolean expression, with the same conventions as in the test in an `\ifthenbox` (so `\n` and `\p` are defined as the current overlay number and the current value of the pause counter). If the test is true, the frame (if `framed=true`) is colored using `fcolor`, the background (if `opaque=true`) is colored using `bgcolor`, and a drop shadow is shown (if `shaded=true`).

Drop shadows may be configured via their own set of style parameters:

```
parameter          default           meaning

shadow.color       foreground        color of shadow
shadow.contrast    100               how dark the shadow appears
shadow.fuzz        100               how much of a fuzzy edge appears
shadow.hoffset     100               horizontal offset of shadow
shadow.voffset     100               vertical offset of shadow
shadow.resolution  100               how "fine grained" is the fuzz drawn
```

5.12 Theorems and such

The command \newtheoremframe creates a new environment that typesets theorems and such in a nice, two-toned frame — inspired by a similar effect in beamer, but without fancy rounded corners. It takes 2 arguments: the environment name and the title. It works just like the basic form of LaTex's \newtheorem.

There is also a command \renewtheoremframe that works the same, but redefines the environment.

For example:

```
\newtheoremframe{claim}{Claim}

\begin{claim} ... \end{claim}   % title reads "Claim"
\begin{claim}[1] ... \end{claim}  % title reads "Claim 1"
\begin{claim}[(key step)] ... \end{claim}  % title reads "Claim (key step)"
```

All of these commands are defined in terms of a generic environment

```
highlight
```

which takes an optional title argument: if no title is present, then the title bar is not shown at all. For example, one could also just write

```
\begin{highlight}[Claim 1] ... \end{highlight}
```

The following style parameters control the highlight environment, and by extension, all theorem frames.

```
parameter                 default              meaning

highlight.title.format    \it\fgcolor{white}   formatting commands for title
highlight.title.bgcolor   blue!60              background color for title
highlight.body.format     \fgcolor{black}      formatting commands for body
highlight.body.bgcolor    blue!20              background color for body
highlight.shaded          false                draw a drop shadow
```

control the colors and fonts of the highlight environment, and by extension, all, theorem frames.

Of course, you can define a specialized environment with its own style parameters. For example, suppose you want lemmas in green instead of blue:

```
\newtheoremframe{lemma-aux}{Lemma}
\newenvironment{lemma}[1][]{%
   \style{highlight.title.bgcolor=green!60,highlight.body.bgcolor=green!20}%
   \begin{lemma-aux}[#1]%
}{%
   \end{lemma-aux}%
}
```

This example illustrates how one can easily exploit the fact that style parameters obey scope rules. It also illustrates how one can set several style parameters using a single `\style` command.

One can similarly define a specialized highlight environment:

```
\newenvironment{myhighlight}[1][\NULL]{%
  \style{highlight.title.bgcolor=green!60,highlight.body.bgcolor=green!20}%
  \begin{highlight}[#1]%
}{%
  \end{highlight}%
}
```

Note that the default value of the argument to the highlight environment is actually the special token `\NULL`.

Note that within a `theoremframe`, or more generally, a `highlight` environment, one may use the overlay mechanism.

Drop shadows can be fine tuned by setting appropriate style parameters, discussed in §5.11.

5.13 The `\StackBackground` command

A number of commands, like `\Colorbox`, are implemented using the `\StackBackground` command, which may be useful in similar contexts. For example, `\Colorbox` is implemented as:

```
\newcommand{\Colorbox}[2]{{%
   \colorlet{PROJ@boxbgcolor}{#1}%
   \StackBackground{PROJ@boxbgcolor}%
   \colorbox{\filtered{PROJ@boxbgcolor}}{#2}%
}}
```

The purpose of `\StackBackground` is to make the overlay (and fade) mechanism work in environments where the background color temporarily changes.

5.14 Labels, equation numbers, and other counter issues

The usual `\label` and `\ref` mechanism will work fine for the most part (although see below for some issues regarding AMS math environments). The `hyperref` package automatically turns these `\ref`'s into hyperlinks as well. The `\ref` will always take you to the first overlay of the slide containing the `\label`.

There are some issues regarding AMS math environments to watch out for. You can write an equation like

```
\begin{equation} \label{L1} ... \end{equation}
```

which will generate an equation with a number, which you can refer to with `\ref{L1}`. This will work fine, even if the equation appears in a slide with overlays.

Now suppose you want an `align*` environment, where each line is revealed one at a time:

```
\begin{align*}
  \pause x & \refresh =  y + z \\
           & \pause = 2w + 3r
\end{align*}
```

Note the use of `\refresh` to extend the effect of the first `\pause` to the second column.

Now suppose we want to add equation numbers or other tags to these equations. Normally, to get equation numbers on each line, you would use the `align` environment instead, and to get custom tags, you would use the `\tag` or `\tag*` commands. Unfortunately, for TeXnical reasons, these mechanisms do not work very well in conjunction with overlays. To correct these problems, the commands `\Tag` and `\TagRef` are provided, and one should consider using these commands consistently throughout for labeling all equations with numbers or other tags.

The command `\Tag` by default generates a tag as an equation number. You can provide your own tag as an optional argument. The command `\TagRef` is used just like `\ref`: it takes a label as an argument; however, if the tag was an equation number, the reference will already have parentheses.

For example, you can generate an equation with a number as follows:

```
\[ \Tag \label{L1} ... \]
```

and the reference `\TagRef{L1}` will look like `(1)`. Alternatively, you can specify your own tag, say `(*)`. Define:

```
\[ \Tag[($*$)] \label{L1} ... \]
```

and the reference `\TagRef{L1}` will look like `(*)`.

Now let us return to the `align*` example above. If you want equation numbers, write:

```
\begin{align*}
  \pause x & \refresh =  y + z \Tag \label{L2} \\
           & \pause = 2w + 3r \Tag \label{L3}
\end{align*}
```

The equation numbers will appear at just the right time, and the references `\TagRef{L2}` and `\TagRef{L3}` will look like `(2)` and `(3)`.

If you want your own tags, say `(*)` and `(**)`, write

```
\begin{align*}
  \pause x & \refresh =  y + z \Tag[($*$)] \label{L2} \\
           & \pause = 2w + 3r \Tag[($**$)] \label{L3}
\end{align*}
```

The tags will appear at just the right time, and the references `\TagRef{L2}` and `\TagRef{L3}` will look like `(*)` and `(**)`.

One technique used to make equation numbers interact well with the overlay mechanism may be more generally useful. There is a command

```
\ResetCounterAtOverlay{<counter name>}
```

which ensures that at the beginning of each overlay of a slide, the counter `<counter name>` is restored to the value it had at the beginning of the slide. So to ensure that equation numbers don't get incremented too much, the command

```
\ResetCounterAtOverlay{equation}
```

is executed when the class is loaded. You may wish to do the same for other counters that need to be reset in this way.

`\ResetCounterAtOverlay` itself is implemented using a more general mechanism that may be useful. Four commands are defined

```
\AtBeginSlide{<code>} \AtBeginOverlay{<code>}
\AtEndSlide{<code>}   \AtEndOverlay{<code>}
```

Using these commands, you can "register" code that is to be executed at the indicated time. Every call simply adds the code to the list of registered code. When a slide environment is entered, all `AtBeginSlide`-code is executed, then for each overlay, the `AtBeginOverlay`-code is executed at the beginning of the overlay, and the `AtEndOverlay`-code is executed at the end of the overlay; finally, after the last overlay, the `AtEndSlide`-code is executed. As an example, `\ResetCounterAtOverlay` is implemented as follows:

```
\newcommand\ResetCounterAtOverlay[1]{%
  \newcounter{PROJ@saved@@#1}%
  \AtBeginSlide{\setcounter{PROJ@saved@@#1}{\value{#1}}}%
  \AtBeginOverlay{\setcounter{#1}{\value{PROJ@saved@@#1}}}%
}
```

5.15 Colored tables

The `colortbl` package is loaded by the `projector` class. You can use this to easily make tables with colored rows and columns.

Consider the following:

```
\pause

\begin{tabular}{cc}
\rowcolor{\filtered{blue}} one &  two \\
\rowcolor{\filtered{green}} three &  four
\end{tabular}
```

Here, \rowcolor is a special command provided by the colortbl package. The use of \filtered will make the entire table (both foreground and background) remain hidden until the right moment, until the appropriate overlay is reached. One can also color columns:

```
\begin{tabular}{
   >{\columncolor{\filtered{red}}}c
   >{\columncolor{\filtered{blue}}}c
}
1 & 2 \\
3 & 4
\end{tabular}
```

Here, \columncolor is a special command provided by the colortbl package. The >{...} construct allows one to insert arbitrary text or commands in each cell of the given column.

However, suppose you want to reveal the table one row or column at a time. This is just a bit tricky, and for convenience, commands \Row, \Col, \Cell, \SaveP, and \ResP are provided to facilitate this.

Here is a table colored by rows and revealed by rows:

```
\begin{tabular}{*{3}{>{\Col}c}}
\pause\Row[red]A &  B & C \\
\Row[green]D & E & F \\
\pause\Row[blue]D & E & F \\
\Row[blue]D & E & F
\end{tabular}
```

The \Row[<color>] command specifies a row color, while the \Col command just assists in keeping each column looking right.

Here is a table colored by columns and revealed by columns:

```
\SaveP
\begin{tabular}{ >{\ResP\pause\Col[red]}c >{\Col[blue]}c
                 >{\pause\Col[yellow]}c }
\Row A &  B & C \\
\Row D & E & F \\
\Row D & E & F
\end{tabular}
```

Here, the \Col[<color>] command specifies the column color. Note the use of \SaveP right before the table, and \ResP in the first >{...} construct. This ensures the pause counter gets reset at the beginning of each row.

Here is a table colored by columns and revealed by rows:

```
\begin{tabular}{ >{\Col[red]}c >{\Col[orange]}c >{\Col[blue]}c }
\pause\Row A &  B & C \\
\Row D & E & F \\
\pause\Row D & E & F \\
\Row D & E & F
\end{tabular}
```

Here is a table colored by rows and revealed by columns:

```
\SaveP
\begin{tabular}{ >{\ResP\pause\Col}c *{2}{>{\pause\Col}c} }
\Row[red]A &  B & C \\
\Row[green]D & E & F \\
\Row[blue]D & E & F \\
\Row[blue]D & E & F
\end{tabular}
\end{center}
```

Here is the same table with one cell colored differently:

```
\SaveP
\begin{tabular}{ >{\ResP\pause\Col}c *{2}{>{\pause\Col}c} }
\Row[red]A &  B & C \\
\Row[green]D & E & F \\
\Row[blue]D & E & F \\
\Row[blue]D & \Cell{Olive}E & F
\end{tabular}
```

Finally, note that if a cell lies in a row with an explicitly specified color, and a column with an explicitly specified color, then the column color always trumps the row color, and an explicit cell color trumps everything. For example:

```
\SaveP
\begin{tabular}{ >{\ResP\pause\Col}c >{\pause\Col[yellow]}c >{\pause\Col}c }
\Row[red]A &  B & C \\
\Row[green]D & E & F \\
\Row[blue]D & E & F \\
\Row[blue]D & \Cell{Olive}E & F
\end{tabular}
```

Here, all of column 2 is yellow, except the last cell in that column, which is Olive.

Instead of using `\pause` commands in the above examples, you can just as well use `\ShowFrom` commands to reveal rows or columns in an arbitrary order. Here is the same table as in the last example, but revealed in reverse column order.

```
\begin{tabular}{ >{\ShowFrom{\p+3}\Col}c
                 >{\ShowFrom{\p+2}\Col[yellow]}c
                 >{\ShowFrom{\p+1}\Col}c }
\Row[red]A &  B & C \\
\Row[green]D & E & F \\
\Row[blue]D & E & F \\
\Row[blue]D & \Cell{Olive}E & F
\end{tabular}
\pause[3]
```

NOTE: these commands work equally well with the tabular* and array environments.

NOTE: The `\Col` command always executed a `\refresh`. Under unusual circumstances, this may give you results other than what you might expect. You can always add a `\record{@}` command just before the table to remedy this.

NOTE: The overlay mechanism will not work directly within a cell. You must use a special command called `\Panel`. For example:

```
\begin{tabular}{*{3}{>{\Col}c}}
\pause\Row[red]A &  B & C \\
\Row[green]D & E & F \\
\pause\Row[blue]D & \Panel{E1 \pause E2} & \Panel{\ShowFrom{\p} F}
\end{tabular}
```

The first `\pause` will reveal rows 1 and 2; the second `\pause` will reveal `D` and `E1` in row 3, and all of row 4; the third `\pause` will reveal `E2` and `F` in row 3.

5.16 Customizing lists

A mechanism is provided for controlling the style and layout of list environments in general, and the itemize and enumerate environments in particular.

The style parameter list.margin controls the indentation of all lists. For example:

```
\style{list.margin=120}
```

sets the indentation to 120% of the default.

The style parameter `list.label.margin` controls the gap between any label and the text. For example:

```
\style{list.label.margin=120}
```

sets the size of the gap to 120

To customize the type of bullets used in the itemize environment, set the style parameters

```
item.label.1, item.label.2, item.label.3, item.label.4
```

which control what is printed at the 1st, 2nd, 3rd, and 4th level of itemize.

For example, the default value of `item.label.1` is `\(bullet\)`. Suppose you want to change it to a red pointing hand.

```
\usepackage{pifont}
\style{item.label.1={\fgcolor{red}\ding{43}}}
```

The package `pifont` provides many funny little symbols that may be useful. See the document "Using common postscript fonts with Latex" for a complete list (see `www.ctan.org/tex-archive/macros/latex/required/psnfss/psnfss2e.pdf`).

The style parameter `item.format` can be set to any command sequence that you want to apply to all itemize labels. To make all itemize labels at all levels red:

```
\style{item.format={\fgcolor{red}}}
```

There are also style parameters for customizing the labels of the enumerate environment. These are:

```
enum.label.1, enum.label.2, enum.label.3, enum.label.4
```

These style parameters are processed in a special way. Suppose you want the enumerate labels at level 1 to look like "1:", "2:", etc. You specify this as follows:

```
\style{enum.label.1={\N\arabic:}}
```

Suppose you want instead "(i)", "(ii)", etc. You specify this as follows:

```
\style{enum.label.1={(\N\roman)}}
```

In general you write whatever combination of text and commands that you like, writing `\N\XXX` (or `\N{\XXX}`) where you want the number to appear. Here, `\XXX` is one of `\arabic`, `\roman`, `\Roman`, `\alph`, or `\Alph`, and specifies how the counter to be encoded.

In fact, the command `\XXX` can be any command that encodes a counter as text. You can define your own encoding command. There is a special command `\DefineSpecialEncoding`, which takes 10 arguments, the first is the name of a new command, and the remaining are the encodings of 1–9. For example:

```
\usepackage{pifont}

\DefineSpecialEncoding{\DING}{\ding{172}}{\ding{173}}{\ding{174}}%
  {\ding{175}}{\ding{176}}{\ding{177}}{\ding{178}}{\ding{179}}{\ding{180}}

\style{enum.label.1={\N\DING}}
```

gives you labels that are little numbers inside circles. The command `\DefineSpecialEncoding` takes care to ensure that all this will work together nicely with the usual `\label` and `\ref` mechanism.

In an enumeration label format string, in addition to the macro `\N`, the macros `\A`, `\B`, and `\C` have special meaning: they refer to the encodings of the counters at levels 1, 2, and 3. For example, suppose you want labels at level 1 to look like "1." and labels at level 2 to look like "1-a:" The following does the trick:

```
\style{enum.label.1={\N\roman.}, enum.label.2={\A-\N\alph:}}
```

The style parameter `enum.format` can be set to any command sequence that you want to apply to all enumerate labels. To make all enumerate labels at all levels red:

```
\style{enum.format={\fgcolor{red}}}
```

Finally, the style parameters

```
enum.sep.1, enum.sep.2, enum.sep.3, enum.sep.4
```

control the formatting of label references (as generated by the `\ref` command). For example:

```
\style{enum.label.1={\N\Roman.}, enum.label.2={\N\Alph.}, enum.sep.2={-}}
```

makes labels at level 1 look like "I.", and labels at level 2 look like "A." A reference to a label at level 2 looks like "I-A". In general, label reference is generated as

$$S_1 E_1 S_2 E_2 \ldots$$

where S_i is defined by `enum.sep.i`, and E_i is the encoding of the counter at level i.

Example. Suppose one wants an itemized list where each bullet gets highlighted with each pause, but all text remains visible. There are a number of ways to do this. Here is one:

```
\begin{itemize} \style{item.format={\alertAt{\p}}, hidden=0}
\pitem a
\pitem b
\pitem c
\pitem d
\end{itemize}
```

If you want, define a new environment:

```
\newenvironment{AlertItemize}{%
  \begin{itemize}%
     \style{item.format={\alertAt{\p}}}%
     \ifvisible{\style{hidden=0}}{}%
}{%
  \end{itemize}%
}
```

This definition is slightly more sophisticated: if the entire itemize list is hidden, then the `\ifvisible` command ensures that it remains hidden. The same can be done with an enumerate list, setting the style parameter `enum.format` instead.

Example. Here we make an itemized list where the bullet changes across overlays, and all text remains visible. Here, the current item has a check mark, previous items a cross, and future items a pencil.

```
\newcommand\SpecialList{%
   \style{hidden=0}%
   \style{item.label.1=%
      \ifthenbox{\n > \p}{\ding{55}}{%
      \ifthenbox{\n = \p}{\ding{51}}{\ding{48}}}%
   }%
}
% note the use of \ifthenbox to prevent "jitters"

...

\begin{itemize} \SpecialList
\pitem a
\pitem b
\end{itemize}
```

5.17 Verbatim text and the `slide*` environment

For TeXnical reasons, the `\verb` command and `verbatim` environment cannot be used in the `slide` environment. If you want to use such "verbatim text" on a slide, use the `slide*` environment instead.

The syntax for the `slide*` environment is the same as for the `slide` environment, but with the following restrictions: (1) the text `\end{slide*}` must appear on a line by itself, possibly with leading and trailing blanks/tabs, and (2) the `slide*` environment may not itself be an argument to any other command.

You may want to consider using the `fancyvrb` package. It provides a number of nice features. Note that if you use line numbering with the `Verbatim` environment from the `fancyvrb` package, you should adjust the `fancyvrb` parameters `xleftmargin` and/or `numbersep`; otherwise, the line numbers will disappear off of the left side of the slide.

You can use the overlay mechanism in a `slide*` environment as usual. With some effort, you can even put pauses and other overlay commands within the verbatim text. Here is a simple recipe for making verbatim boxes with arbitrary LaTex commands, using the `fancyvrb` package.

In the preamble:

```
\usepackage{fancyvrb}

\newcommand\BackSlash{\char92}
\newcommand\LeftBrace{\char123}
\newcommand\RightBrace{\char125}

\DefineVerbatimEnvironment{Vrb}{Verbatim}{commandchars=\\\{\}}
\newenvironment{Quote}{\let\\\BackSlash\let\{\LeftBrace\let\}\RightBrace}{}
```

And then:

```
\begin{slide*}

\begin{Quote}
\begin{Vrb}
\fgcolor{red}\\foo\{bar\}
\pause{}Xfoo/bar-
\refresh{}Xfoo/bar-
\end{Vrb}
\end{Quote}

\end{slide*}
```

Notes:

- With this recipe, the characters \, {, and } have their usual meaning, so that you can invoke arbitrary LaTex commands.
- \\, \{, and \} are written to get the literals \, {, and }.
- Each end of line in a verbatim box closes scope, so coloring and overlay effects do not extend beyond the end of line (use `\refresh` if necessary).

How it works. The lines between `\begin{slide*}` and `\end{slide*}` are written *verbatim* to a temporary file `\jobname.vrb` (e.g., if the LaTeX input file is `foo.tex`, the temporary file is `foo.vrb`). In fact, this is done using the same techniques used to implement verbatim environments. Then the usual `slide` environment is invoked, with `\input{\jobname.vrb}` as its body.

Using `slide*` *for debugging.* When using the `slide` environment, information about input line numbers is lost, and any LaTeX error messages will not give meaningful line numbers. If you have difficulty tracking down an error, using the `slide*` environment should give you an error message with the line number within the temporary file `\jobname.vrb`, which will help pinpoint the error.

5.18 Other issues

\parbox's have some "issues". In the implementation of \parbox, various layout settings are restored to "default" values which are undesirable: the "ragged right" setting is lost, as well as some the vertical skip settings, like \parskip.

To correct these problems, add the command \fixparbox:

```
\parbox{0.5\textwidth}{\fixparbox ... }
```

Fortunately, this problem does not exist for minipage environments, because LaTex provides a "hook" to insert formatting code into the minipage, which the projector class exploits — unfortunately, there is no such hook for \parbox's.

TIP: available documentation often does not mention the extended capabilities of \parbox and minipage. One can write:

```
\parbox[pos][height][innerpos]{width}{...text...}
\begin{minipage}[pos][height][innerpos]{width} ...text... \end{minipage}
```

where pos and innerpos are in {t,b,c,s}, and control the external and internal vertical alignment of the resulting box.

Another problem with both \parbox's, minipage's, and \vbox's in general, is that you may under some circumstances get an unwanted paragraph skip at the beginning. This will happen if you issue any \vspace command, or any \color command (and by extension, and overlay command, like \pause), at the beginning of the box.

One fix is to place the command

```
\vspace{-\parskip}
```

at the beginning of the box to avoid this problem — although external alignment may be adversly affected.

If the first command is a \color command, one can try moving the command out of the \parbox, or alternatively, to leave it in the \parbox, but precede it immediately (no spaces) with the command \mbox{}.

6 Summary

6.1 Commands and environments

```
.....................................
  n = overlay number
  i = integer
  c = color name or expression
  t = any text or command sequence
  l = length
  % = percentage
.....................................
```

```
\begin{slide} ... \end{slide}
\begin{slide*} ... \end{slide*}

\pause[i],              \PauseStep[i],       \PauseSet[i]
\pitem
\visible
\invisible
\hideFrom{n},           \hideTo{n},           \hideAt{n}
\showFrom{n},           \showTo{n},           \showAt{n}
\HideFrom{n},           \HideTo{n},           \HideAt{n}
\ShowFrom{n},           \ShowTo{n},           \ShowAt{n}
\colorFrom{n}{c},       \colorTo{n}{c},       \colorAt{n}{c}
\alertFrom{n},          \alertTo{n},          \alertAt{n}
\fadeFrom{n}{%},        \fadeTo{n}{%},        \fadeAt{n}{%}
\refresh[t]
\record{t}
\trimslide[i]

\alert
\fade{%}
\fgcolor{c}
\bgcolor{c}

\Colorbox{c}{t}
\FColorbox{c}{c}{t}

\filtered{c}

\ifvisible{t}{t}

\upbox[l]{t}, \downbox[l]{t}, \centerbox[l]{t}

\ifthenbox[<position>]{<test>}{t}{t}
\graphicbox[<includegraphics options>]{<file name>}

\putbox{l}{l}{t}

\overlaysoff
\ifoverlaysoff{t}{t}

\squeeze

\banner{t}, \Banner{t}, \BANNER{t}, \Stripe{t}
```

```
\begin{BOX}[<parameter settings>] ... \end{BOX}

\newtheoremframe{t}{t}
\renewtheoremframe{t}{t}

\begin{highlight}[t] ... \end{highlight}

\fadeamount
\lastslide

\targetslide[n]{<target name>}

\Tag[t]
\TagRef{<label>}

\ResetCounterAtOverlay{<counter name>}

\Row[c], \Col[c], \Cell{c}, \SaveP, \ResP, \Panel{t}

\AtBeginSlide{t}
\AtBeginOverlay{t}
\AtEndSlide{t}
\AtEndOverlay{t}

\DefineSpecialEncoding{<command name>}{t1}{t2} ... {t9}

\fixparbox

\dimdiv{<macro name>}{l}{l}
```

6.2 Lengths

```
\smargin
\tmargin
\bmargin
\fsmargin
\ftmargin
\fbmargin
\footwidth
```

These should not be directly modified.

6.3 Counters

```
slidenum
overlaynum
pausectr
```

These should normally not be directly modified.

6.4 Booleans

```
visibleP
overlaysoffP
```

These should not be directly modified.

6.5 Color names

```
foreground   -- initially white
background   -- initially black
```

These should normally not be directly modified.

6.6 Style parameters

Style parameters can be set with the command

```
\style{<name1>=<value1>, <name2>=<value2>, ...}
```

The `\style` command is implemented using the `keyval` package, which is loaded by the `projector` class. As such, you can insert spaces around the `=` and `,` characters, and you can enclose values in brackets (`{` and `}`) if convenient.

There is a lower-level function called `\setparam`. `\setparam{#1}{#2}` is essentially equivalent to `\style{#1={#2}}`.

Also, for style parameters that take a percentage as a value, you can use the command

```
\scaleparam{<name>}{x}
```

to multiply the current value of `<name>` by the real number `x`. For example,

```
\scaleparam{vskip}{120}
```

sets the value of the style parameter `vskip` to 120, while

```
\style{vskip=1.2}
```

multiplies the current value of vskip by 1.2.

The effects of \style, \setparam, and \scaleparam extend to the end of the enclosing scope. You may reset a style parameter at any time, except for those named slide..., which should only be set in the preamble.

The command \paramval{<name>} expands to the current value of <name>.

The following is a list of all style parameters, with their type and default values. The types are:

```
  % = percentage of default
  c = color
  b = boolean
  t = arbitrary text or commands
  f = special enumerate label format string
  l = length
```

Name	Type	Default
slide.margin	%	100
slide.margin.side	%	100
slide.margin.bot	%	100
slide.margin.top	%	100
slide.size	%	100
slide.aspect.ratio	<ratio>	4:3
vskip	%	100
hidden	%	100
alert.color	c	blue!80!black
foot.margin	%	100
foot.margin.side	%	100
foot.margin.bot	%	100
foot.margin.top	%	100
foot.text.top.left	t	
foot.text.top.center	t	
foot.text.top.right	t	
foot.text.bot.left	t	
foot.text.bot.center	t	
foot.text.bot.right	t	
foot.format	t	

```
banner.format               t    \large\fgcolor{blue!80!black}
Banner.format               t    \large\bf\fgcolor{blue!80!black}
BANNER.format               t    \Large\bf\fgcolor{blue!80!black}

banner.env                  t    center
Banner.env                  t    center
BANNER.env                  t    center

banner.aboveskip            l    0pt
Banner.aboveskip            l    0pt
BANNER.aboveskip            l    0pt

banner.belowskip            l    \smallskipamount
Banner.belowskip            l    \medskipamount
BANNER.belowskip            l    \bigskipamount

Stripe.margin               %    100
Stripe.bgcolor              c    blue!60
Stripe.env                  t    center
Stripe.format               t    \bf\large\fgcolor{white}

BOX.fwidth                  %    100
BOX.margin                  %    100
BOX.bgcolor                 c    blue!20
BOX.fcolor                  c    foreground
BOX.framed                  b    false
BOX.opaque                  b    false
BOX.shaded                  b    false
BOX.extended                b    false
BOX.when                 <test>  \boolean{true}

shadow.color                c    foreground
shadow.contrast             %    100
shadow.fuzz                 %    100
shadow.hoffset              %    100
shadow.voffset              %    100
shadow.resolution           %    100

highlight.title.format      t    \it\fgcolor{white}
highlight.title.bgcolor     c    blue!60
highlight.body.format       t    \fgcolor{black}
highlight.body.bgcolor      c    blue!20
highlight.body.shaded       b    false
```

```
list.margin                %    100
list.label.margin          %    100

item.label.1               t    \(\bullet\)
item.label.2               t    \(\circ\)
item.label.3               t    \(\cdot\)
item.label.4               t    -

item.format                t

enum.label.1               f    \N\arabic.
enum.label.2               f    \N\alph.
enum.label.3               f    \N\roman)
enum.label.4               f    \N\Alph.

enum.format                t

enum.sep.1                 t
enum.sep.2                 t
enum.sep.3                 t    .
enum.sep.4                 t    )
```

Note that for types `t`, `l`, `f`, and `<test>`, the `\style` command uses `\def` to assign values; for other types, it uses `\edef`, meaning that the value to be assigned to the parameter is evaluated immediately at the time `\paramval` is executed.

Also note that when setting boolean parameters with the `\style` command, writing `name` is the same as writing `name=true`.

6.7 Preloaded packages

These LaTex packages are loaded in the given order:

```
amsmath
latexsym
url
calc
ifthen
xcolor (loaded with options x11names and svgnames)
colortbl
graphicx
fancyhdr
hyperref
fp (loaded with option nomessages)
keyval
```

6.8 Change Log

Version 2.4 (Jan. 15, 2008)

- added `extended` style parameter for the `BOX` environment

Version 2.3 (Sept. 12, 2008)

- fixed some internal names

Version 2.2 (Sept. 11, 2008)

- added `\Stripe` command

Version 2.1 (May 9, 2008)

- some bug fixes

Version 2.0 (May 8, 2008)

- added drop shadows
- redefined `slide*` (semantics have changed to allow overlays)
- added `BOX` environment for very fancy colored boxes
- added `\style` command for syntactically simpler assignment to style params
- changed typesetting semantics (slightly) of headers and footers
- added `\Panel` command that makes the overlay mechanism work in a `colortbl` environment
- modified semantics (slightly) of `\fade` command
- modified `\targetslide` semantics
- introduced `edef` option of style parameters, to force immediate evaluation
- added command `\dimdiv`, which may be useful in scaling certain lengths
- name change: `\hmargin` → `\smargin`, `\fhmargin` → `\fsmargin`
- packages `fp` and `keyval` loaded
- revised documentation

Version 1.5 (March 21, 2008):

- simplified implementation of \upbox, \downbox, \centerbox, \ifthenbox

Version 1.4 (March 21, 2008):

- added commands \upbox, \downbox, \centerbox

Version 1.3 (March 20, 2008):

- added commands \Colorbox, \FColorbox
- modified the color-mixing logic of the \fade command slightly

Version 1.2 (March 2, 2008):

- added commands \fade, \fadeFrom, \fadeTo, \fadeAt

Version 1.1 (Jan. 17, 2008):

- do not load packages amssymb and amsfonts (these can cause conflicts with certain custom font packages, and are not needed by the projector class itself)
- do not load the amstext package (already loaded by amsmath)

Version 1.0 (Nov. 6, 2007)