LaTeX

powerdot — making presentations with LaTeX

Hendri Adriaens and Christopher Ellison

Abstract

This article describes the powerdot class [2], for making presentations with LaTeX. It is a successor to the prosper and HA-prosper packages.

1 Introduction

powerdot is a presentation class for LaTeX that allows for the quick and easy development of professional presentations. It comes with many tools that enhance presentations and aid the presenter. Examples are automatic overlays, personal notes, a digital clock on slides and a handout mode. To view a presentation, DVI, PS or PDF output can be used. A powerful template and palette system is available to easily develop new styles. Also, a LyX layout file is provided. powerdot is a new package in the line of prosper [5] and HA-prosper [1].

It has been well known for quite some time that the prosper class has severe problems. Examples include damaged constructions from a redefined `\item`, spacing problems on overlays while in math mode, failing counter protection, useless DVI and PS output, and a lack of support for screen-optimized paper dimensions. The HA-prosper package (developed by the first author) tried to correct some of these problems, but with additional LaTeX programming experience, it was found that some of the problems of the prosper class (such as the paper dimensions) could not be corrected.

As an alternative, the idea of using pstricks [6, 7] and `minipage` environments for content was appealing in that it allowed for a vast variety of presentation styles.

Halfway through 2004, Hendri decided to make a successor to the prosper and HA-prosper combination. The class would be built from the ground up, and it would be called powerdot.[1] As it would be a major undertaking to develop a new class, new styles, and documentation, Hendri looked for a helping hand on the HA-prosper mailing list. He was very lucky to find that Chris Ellison was prepared to help. After some initial tests, the production of the class finally started in July 2005, and it was mostly completed during the summer holidays of 2005. This article describes the build process and the choices made along the way.

2 Paper size and orientation

Before generating output, we needed to be sure we were using the correct paper size and orientation. Our general idea was to place all content in `minipage` environments and then use pstricks' `\rput` to position the environments on the paper. Therefore, powerdot itself could control page dimensions and margins for the user. So, we removed all margins and defined the origin `(0,0)` at the lower-left corner of the paper and `(\slidewidth,\slideheight)` for the upper-right corner. This provides an easy way for designers to create scalable styles for use with multiple paper types, e.g., letter paper, A4 paper and screen ratio "paper" (4/3).

But what are these lengths `\slidewidth` and `\slideheight`? They are determined from the paper type and orientation specified by the user and will be set to `.5\paperwidth` and `.5\paperheight`. We then magnify the DVI by a factor of two to have easy access to large fonts with standard files such as `size10.clo`. This creates a usable DVI file,[2] a usable PS file (after processing with dvips), and a usable PDF file (after processing with ps2pdf).

To help the user when compiling to PDF, powerdot uses the `papersize` special to tell dvips which paper size should be used. This way, the user need not specify the paper type with dvips' `-t` command line option. Unfortunately, there is a problem with this special. Most dvips configurations used today have a paper name `A4size` which, when A4 paper dimensions are found in the `papersize` special, does not write the PostScript `a4` command to the PostScript file. When processing this PostScript file using ps2pdf without command line parameters, the program will not find a particular paper type and will default to letter paper. To avoid this problem, powerdot explicitly writes the `a4` command to the PostScript file when A4 paper is requested, and the `letter` command for letter paper.[3]

3 Designer interface

So far, we have set up the paper dimensions and

Editor's note: This article was also published in *MAPS* 33 (Najaar 2005), pp. 54–58, and is published here, with additions, by kind permission of the author and editor.

[1] At first, the name TeXciting was chosen, but that was abandoned due to associations with 'citations'.

[2] For DVI viewers that understand PostScript `\special`s.

[3] powerdot also has the `nopsheader` option, which avoids writing the `papersize` special and the `a4` command. This should be used when dvips can't be used without command line parameters; for instance, when the editor always inserts either `-tletter` or `-ta4`.

made sure that the user can get a proper DVI, PS or PDF file without much trouble or knowing about command line parameters. Now we have to make sure that new slide styles can easily be developed. This will be a huge improvement over prosper's complicated and basically absent designer interface.

Remember, we started with the idea of placing content on the paper in `minipage` environments using `\rput`. This gives rise to a very simple but powerful designer interface where all properties of the main components (slide title, text box, etc.) can be controlled by keys (options), which are defined using xkeyval [3]. These keys can be used in powerdot's `\pddefinetemplate` command, which has another argument to create the background of the slide (using, for instance, pstricks). A special key, called `ifsetup`, can be used to specify to which setups all following keys should apply. For instance,

```
ifsetup={landscape,a4paper}
```

tells powerdot that all following keys should be used if the user requested landscape A4 paper. The following, however,

```
ifsetup=landscape
```

makes all following keys be used in landscape orientation with any paper type. powerdot also provides a `\pdifsetup` command that works in a similar way as the key, but takes true and false texts, executing one of them depending on the current setup of the document and the first argument, which is like the input to the `ifsetup` key.

The `\pddefinetemplate` command allows us to use an existing template as the basis for a new template, which further simplifies style development. Here is an example of the designer interface.

```
\documentclass[
  % orient=portrait
]{powerdot}
\pddefinetemplate{basic}{
  titlepos={.05\slidewidth,.91\slideheight},
  titlewidth=.9\slidewidth,
  textpos={.05\slidewidth,.85\slideheight},
  textwidth=.9\slidewidth,
  textfont=\raggedright\color{black}
}{%
  \psframe*[linecolor=yellow!20]%
  (0,0)(\slidewidth,\slideheight)%
}
\pddefinetemplate[basic]{slide}{%
  ifsetup=landscape,
  titlefont=\Large\raggedright\color{black},
  ifsetup=portrait,
  titlefont=\Large\centering\color{black}
}{}
\begin{document}
  \begin{slide}{Title}
    Some text.
  \end{slide}
\end{document}
```

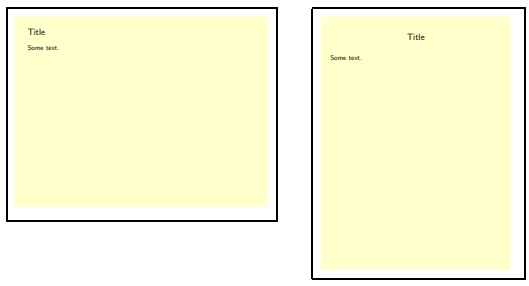
The first `\pddefinetemplate` command creates a template named `basic`, which defines the title and text position on the page, and (in the second argument) the background of the slides (here, a light yellow color).

The second `\pddefinetemplate` command defines a `slide` template, in this case based on the `basic` template. This template initializes the position of the main text box and the title and the text font to be used. In addition to the declarations coming from the `basic` template, the `slide` template specifies the title formatting (font, justification, and color).

Here we use the `ifsetup` key to choose different formatting for the slide title in landscape mode (`\raggedright`) or portrait mode (`\centering`). In practice, this might be considered inconsistent design, but here it just serves as an example. This example is simple, and the templates could easily be merged into one, but it clearly demonstrates the possibility of reusing existing templates.

Finally, we actually produce an example slide, using the just-defined `slide` environment.

If we typeset the example above in both landscape and portrait orientation, we get the following output.



When a designer wants to do more fancy things which cannot be controlled by keys, powerdot supplies a variety of macros that do specific jobs and can be redefined to achieve any desired goals. Examples are `\pd@title`, which controls the typesetting of the presentation title, and `\pd@slidetitle`, which controls the typesetting of slide titles. By default, these macros just pass on their argument, but they can be redefined to do arbitrary things.

As examples of the possibilities of the design interface of powerdot, you can find samples of some

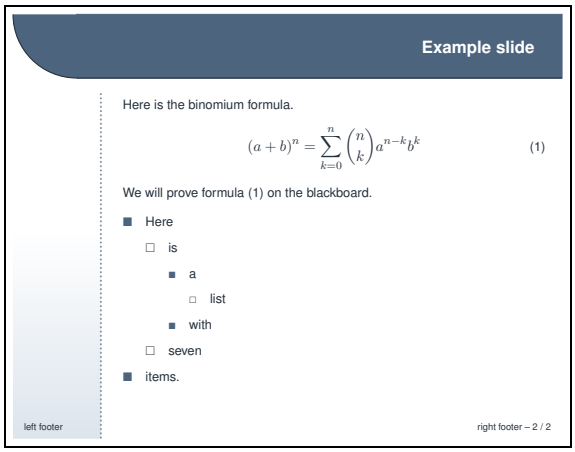


Figure 1: sailor style

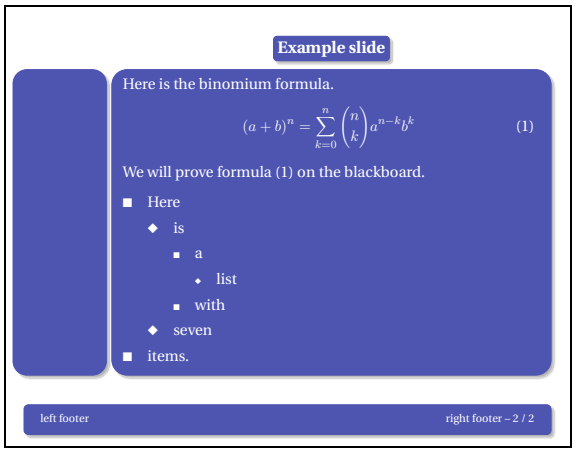


Figure 2: bframe style

of the currently available presentation styles in figures 1 to 3.

4 User interface

Most importantly, a new user interface needed to be developed which was both powerful and simple to use. Setting up the main characteristics of a presentation, like paper type, font size and style, is done via the `\documentclass` command. Other settings, like the footers, transition effects and layout of lists, is done via the `\pdsetup` command.

The user interface for making slides is intended to be very simple and is mainly formed by the `slide` environment.[4] By default, this environment first stores the literal text of the body in a token register. This allows us to reuse the body later on. We do this by searching the input stream for the

[4] Most styles supply additional templates, such as the `wideslide` environment, but these work internally the same as the `slide` environment.

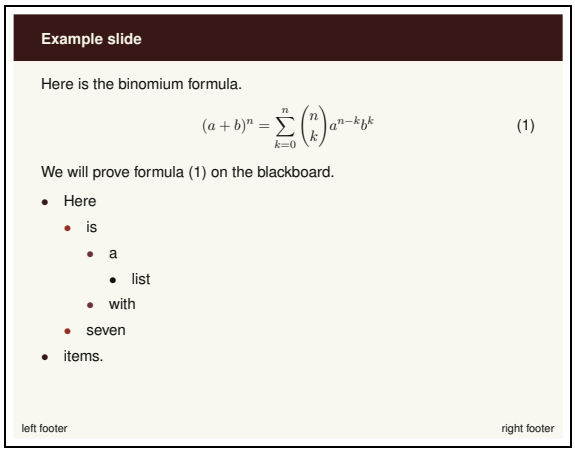


Figure 3: paintings style

next occurrence of the `\end` command. If this command has the proper argument, namely `slide`, then we have found the end of the slide and we can start processing the content. If not, we add the text found so far to the token register and continue the search.

Now that we have the body 'in our hands', we can typeset it once and see what happens. The user could actually have specified an overlay command like `\onslide` or `\pause` in the slide. During the first run, these commands are executed and these are used to determine the remaining number of times that we need to typeset the body. This process creates several overlays using just one slide environment. Here is an example.[5]

```
\begin{slide}{My first slide}
  Hello \pause world!
\end{slide}
\begin{slide}{My second slide}
  \onslide{1-}{Hello} \onslide{2}{world!}
\end{slide}
```

This example creates two overlays for each slide. `Hello` will appear on both overlays for each slide, while `world!` appears only on every second overlay.

There is a TEXnical drawback to using the technique described above to get the body of the environment, and that is that the category codes will be fixed in the text once we typeset it for the first time. Hence, constructions that rely on changing catcodes internally, such as the `verbatim` environment, do not work inside the slide environment. Thus, powerdot implements two other techniques to process slides.

The second technique (accessed by the slide option `method=direct`) directly typesets the body of

[5] Please refer to the documentation for syntax details.

the slide, instead of storing it first in a token register. This is fast, and allows for verbatim listings on slides, but doesn't allow for overlays.

The third technique (accessed by the slide option `method=file`) writes the body of the slide to a temporary file. This file can be read back in again to produce the slide. This method does allow for verbatim on slides and for overlays. However, since an external file is needed, this is a little bit slower than the other two methods.

Here is an example for having both verbatim and overlays on slides.

```
\begin{slide}[method=file]{Verbatim and
                           overlays}
 \begin{lstlisting}[frame=single,
                    escapeinside=‘’]
   the first line of code‘\pause’
   the second line of code‘\pause’
   the third line of code
 \end{lstlisting}
\end{slide}
```

The example uses the listings package and creates three overlays on which the program listing is revealed step by step.

5 Supporting LaTeX commands

Of course, creating a presentation is rather different from writing an article, and by introducing new features, such as overlays, we might bring trouble to standard LaTeX constructions.

LaTeX counters are one example. When repeatedly typesetting the same text, a counter increase in that text (for instance by the `equation` environment) gets executed each time. This could lead to the same equation having different numbers on different overlays. This is easily overcome, however. We record the value of known counters before typesetting the first overlay and reset it at the start of the next overlay. powerdot does this automatically for the counters `equation`, `figure`, `table` and `footnote`. The user can add more counters to the list by using the `counters` key in the `\pdsetup` command.

A similar example is the `\label` command. If the standard `\label` command were executed on overlays, the user would always get errors about `Multiply defined labels`. prosper tried to solve this issue by executing `\label`s only on the first overlay. It is obvious that this leads to undefined labels when a label does not appear on the first overlay, for instance, because it was gobbled by, for example, `\onlySlide*{2}{...}`. Another idea would be to tell the user to always use `\label` inside an appropriate `\onslide` command with a single overlay specification to avoid multiply defined labels. That, however, requires extra work from the user.

In contrast, powerdot executes the `\label` only on the first overlay *where it is actually used*. This could, for example, be overlay 37. The way it does this is by adding all labels defined on a slide to a list. If the list already includes the current label, this label is not executed again. The list is emptied at the start of every slide. The side effect of this system is that multiply-defined labels on the same slide cannot be detected anymore. However, multiply-defined labels on different slides still result in a warning in the log file of the user. This side effect is not considered very serious, as the source of a single slide is usually rather short and errors can be observed in the output.

6 LyX support

To support the use of LyX [4] for creating powerdot presentations, we wanted the user interface to work within the restrictions set by LyX. One of the difficulties with LyX's interface is that it doesn't allow environments to have arguments. Instead, we have to use commands to indicate the beginning and end of a slide. When a powerdot LyX presentation is exported to LaTeX it looks like this:

```
\documentclass{powerdot}
\begin{document}
\lyxend\lyxslide{My first slide}
  Hello \pause world!
\lyxend\lyxslide{My second slide}
  \onslide{1-}{Hello} \onslide{2}{World}
\lyxend
\end{document}
```

Here, `\lyxend` is a harmless macro that is only used by `\lyxslide` as a delimiter. This interface can be extended via the `\pddefinelyxtemplate` command if a style defines custom templates. This command defines a control sequence that uses the underlying templates, like `\lyxslide` uses the `slide` template.

The LyX interface of powerdot also allows for the `direct` and `file` processing methods described in section 4. This does lead to a tricky situation when writing the body of a slide verbatim to a file, because we read material line by line. When seeing `\lyxend`, we need to stop reading verbatim, but as the next slide starts again at the same line, this will also be read verbatim. To be able to execute the next slide again, we also need to write the remainder of the line to a temporary file and read it back in. ε-TeX's `\scantokens` could also be used to do this job, but it has the habit of inserting an end-of-file

into the input stream, which causes trouble if the next slide starts verbatim reading again. This can be patched, but the easier solution of using a physical external file and reading back in exactly one line — ignoring the EOF on the next line — was preferred.

7 Hiding material

How do `\onslide` and `\pause` actually work when hiding material?[6] This is done using the overlays offered by pstricks. We can use this system in the following way. On every slide, we initialize PostScript overlay 0. On that overlay, text will be visible. PostScript overlay 1 is used to make material invisible. This means that it will be typeset as usual by LaTeX, but that the material will not be visible in the output. Hence, the cursor will still be moved by the material. By switching to overlay 1 and back at the right times, we can hide any material we want. By switching to overlay 1 and not switching back, we can hide all following material.

If we consider the example again and ignore all second (powerdot) overlays (as all material will be visible there), in essence it comes down to executing the following:

```
\documentclass{powerdot}
\begin{document}
\makeatletter
\begin{slide}{My first slide}
  Hello \pst@Verb{(1) BOL} world!
\end{slide}
\begin{slide}{My second slide}
  Hello \pst@Verb{(1) BOL}world!%
  \pst@Verb{(0) BOL}
\end{slide}
\end{document}
```

The `\pst@Verb` commands insert the switches to PostScript overlay 0 and 1 into the PostScript document via `\special`'s. We see that `\pause` will not return to overlay 0 afterwards, whereas `\onslide` does so. Hence, any following material would be invisible on powerdot overlay 2 on the first slide and not on the second.

[6] There are also versions of these macros that ignore material or color it with another color than the text color.

8 Final details

The user interface has many additional details — to create sections, table of contents entries, prevent `figure` and `table` environments from floating, create personal notes and handouts, and much more.

Please have a look at the user documentation if you are interested in learning more about the powerdot class. The result of this holiday effort is a class that can create good-looking slides with a minimal amount of input from the designer and user, both when typing the source and when compiling it.

References

[1] Hendri Adriaens. HA-prosper package. CTAN:/macros/latex/contrib/HA-prosper.

[2] Hendri Adriaens and Christopher Ellison. powerdot class. CTAN:/macros/latex/contrib/powerdot.

[3] Hendri Adriaens and Uwe Kern. xkeyval — new developments and mechanisms in key processing. *TUGboat*, 25(2):194–199, 2004. CTAN:/macros/latex/contrib/xkeyval.

[4] LyX crew. LyX website. http://www.lyx.org.

[5] Frédéric Goualard and Peter Møller Neergaard. prosper class. CTAN:/macros/latex/contrib/prosper.

[6] Herbert Voß. PSTricks website. http://pstricks.tug.org.

[7] Timothy Van Zandt et al. PSTricks package, v1.07, 2005/05/06. CTAN:/graphics/pstricks.

◇ Hendri Adriaens
hendri[at]uvt.nl

◇ Christopher Ellison
chris.ellison[at]gmail.com