powerdot
making presentations with LaTeX

Abstract
This article describes some technical details of the powerdot class [3] which was developed during the summer holidays of 2005.

Keywords
LaTeX presentations, powerdot, prosper, pstricks, xkeyval

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 and a handout mode. To view a presentation, DVI, PS or PDF output can be used. A powerful template 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 tried to correct some of these problems, but as LaTeX programming experience grew, it was found that some of the problems of the prosper class (such as the paper job) could not be corrected anymore.

However, 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-proper 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 will describe the build process and the choices made along the way.

Paper size and orientation

Before putting anything on the paper, we needed to be sure we were using the correct paper size and orientation. However, as the idea was to place all content in `minipage` environments and then use pstricks' `\rput` to position the environments on the paper, we really didn't need to worry about page dimensions and margins for the user. So, we removed all margins and placed the origin `(0,0)` in the lower-left corner of the paper and `(\slidewidth,\slideheight)` in the upper-right corner. This provided an easy way for designers to create scalable styles for use with multiple paper types such as letter paper, a4 paper and screen ratio paper (4/3).

But what are these lengths `\slidewidth` and `\slideheight`? They will be 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 the regular files `size10.clo` etcetera. This creates a useable DVI file,[2] a useable PS file (after processing with dvips), and a useable PDF file (after processing with ps2pdf).

To help the user when compiling to PDF, powerdot uses the `papersize` special to tell dvips which paper should be used. This way, the user does not need to specify the paper type with the `-t` command line option. However, there is a problem with this special. Most dvips configurations used today have a special `A4size` paper which, when a4 paper dimension 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.[3,4]

Designer interface

So far, we have set up the paper dimensions and 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 putting 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, etcetera) can be controlled by keys which are defined using xkeyval [2]. These keys can be used in the `\pddefinetemplate` command, which has another argument that creates 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 in case the user requested landscape a4 paper. The following, however,

```
ifsetup=landscape
```

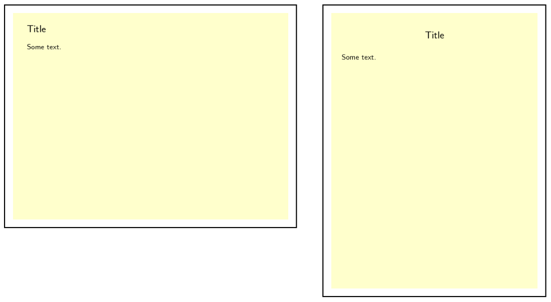
makes all following keys used in landscape orientation, but with any paper type. There is also a 'stand-alone' version of this key called `\pdifsetup`.

Finally, the `\pddefinetemplate` command allows us to use an existing template as basis for a new template, which further simplifies style development. Here is a simple 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 `\pddefinetemplate` command defines the `slide` template which will be 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 font. Note the use of the `ifsetup` key to choose different formatting for the slide title in landscape mode or portrait mode. In practice, this might considered to be inconsistent design, but here it just serves as an example. This example is very simple and the templates could easily be merged into one, but it clearly demonstrates the possibilities to reuse existing templates.

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



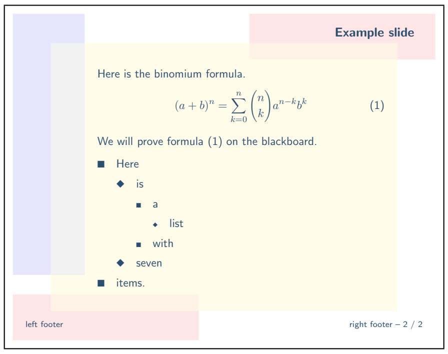
When a designer wants to do more fancy things which cannot be controlled by keys, powerdot supplies access to a variety of macros that do specific jobs and can be redefined to achieve these goals. Examples are `\pd@title`, which controls the typesetting of the presentation title, and `\pd@slidetitle`,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

As an example of the various possibilities of the design interface of powerdot, you can find samples of some of the currently available presentation styles in figures 1 to 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 kept very simple and is mainly formed by the `slide` environment.[5] This environment first stores the literal text of the body in a token register. This allows us to reuse the


Example slide

Here is the binomium formula.

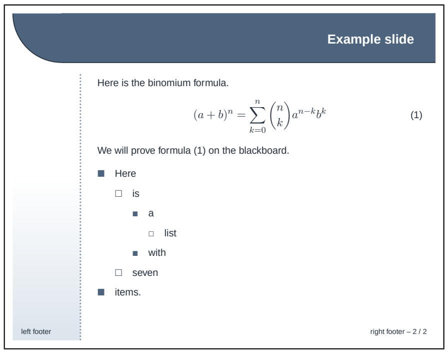
$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k \qquad (1)$$

We will prove formula (1) on the blackboard.

- Here
  - is
    - a
      - list
    - with
  - seven
- items.

left footer right footer – 2 / 2

Figure 1. elcolors style


Example slide

Here is the binomium formula.

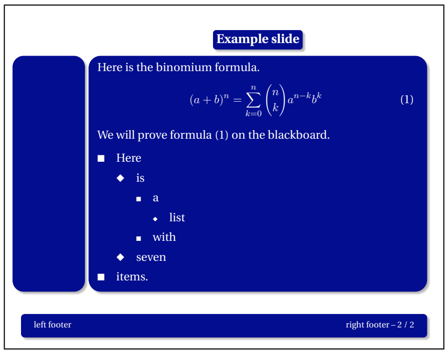
$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k \qquad (1)$$

We will prove formula (1) on the blackboard.

- Here
  - is
    - a
      - list
    - with
  - seven
- items.

left footer right footer – 2 / 2

Figure 2. sailor style


Example slide

Here is the binomium formula.

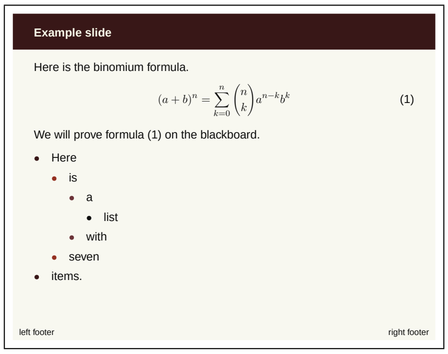
$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k \qquad (1)$$

We will prove formula (1) on the blackboard.

- Here
  - is
    - a
      - list
    - with
  - seven
- items.

left footer right footer – 2 / 2

Figure 3. bframe style


Example slide

Here is the binomium formula.

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k \qquad (1)$$

We will prove formula (1) on the blackboard.

- Here
  - is
    - a
      - list
    - with
  - seven
- items.

left footer right footer

Figure 4. paintings style

body later on. We do this by searching the input stream for the 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 a simple example.[6]

```
\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 drawback to using the technique described above to get the body of the environment and that is that category codes will be fixed in the text once we start typesetting it for the first time. Hence, constructions that rely on changing catcodes internally, like the `verbatim` environment, do not work inside the slide environment. This problem is easily worked around by storing the verbatim text outside the slide environment in a box and using that box inside the slide.

Supporting LaTeX commands

Of course, making a presentation is rather different from writing the article itself and by introducing new features, such as overlays, we might bring standard LaTeX constructions in trouble. Take for instance LaTeX counters. When repeatedly typesetting the same text, counter increases in that text (for instance by the equation environment) also get executed several times by the same text. This could lead to the same equation having different numbers on different overlays. This is easily overcome, however. We just record the value of some 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`, and `table`. The user can easily add more counters to the list by using the `counters` key in the `\pdsetup` command.

Another example is the `\label` command. If the standard `\label` command would be executed on overlays, the user would always get `Multiply defined labels` errors.

prosper tried to solve this issue by executing `\labels` only on the first overlay. The error is obvious. Another idea would be to tell the user to always use `\label` inside an appropriate `\onslide` command with a single overlay specification. That, however, requires extra work from the user.

powerdot executes the `\label` only on the first overlay *where it is actually used*. This could, in fact, 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 slide is often rather short.

LyX support

To support the use of LyX [4] for creating powerdot presentations, the user interface should be able to deal with 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

```
\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 easily be extended by using the `\pddefinelyxtemplate` command in case a style defines custom templates. This command defines a control sequence that uses the underlying templates, like `\lyxslide` uses the `slide` template.

Hiding material

How do `\onslide` and `\pause` actually work when hiding material?[7] This is done using 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 visual 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 back anymore, 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), it comes basically 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}
\makeatother
\end{document}
```

The `\pst@Verb` commands enter 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.

Final details

The final task for the user interface was to fill in the details. An interface was necessary 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.

The future

The development of powerdot will of course continue and we have the plan to add new and innovative features to the class that will be very useful. When writing this, we are also working hard on, amongst many other things, multiple palettes per style (meaning the same design but different colors), a digital clock on slides and random elements (like dots and rings) to make presentations a little livelier. When you read this, powerdot might already have had an update in which all these features, and more, are present.

References

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

[2] Hendri Adriaens. xkeyval package. `CTAN:/macros/latex/contrib/xkeyval`.

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

[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`.

Notes

1. At first, the name TEXciting was chosen, but that was abandoned due to associations with ‘citations’.
2. For DVI viewers that understand PostScript `\specials`.
3. And the `letter` command for letter paper.
4. There is the `nopsheader` option which avoids writing the `papersize` special and the `a4` command. This should be used when the user can't use dvips without command line parameters, for instance, because the editor always inserts either `-tletter` or `-ta4`.
5. Most styles supply more templates, like the `wideslide` environment, but these work internally the same as the `slide` environment.
6. Please refer to the documentation for syntax details.
7. There are also versions of these macros that eat material or color it with another color than the text color.

Hendri Adriaens
`hendri[at]uvt.nl`

Chris Ellison
`chris.ellison[at]gmail.com`