Random Determinism

http://randomdeter-minism.wordpress.com

A style file for eink readers

Recently I bought an Amazon Kindle touch. It is more convenient than the IREX DR1000 for reading morning news and blogs (thanks to ***instapaper's*** automated delivery of "Read Later" articles, and ***ifttt*** for sending RSS feeds to Instapaper).

I have also started reading novels on the Kindle as opposed to the DR1000. Being small, the Kindle is easier to carry; and its hardware just works better than the DR1000: instant startup, huge battery life, and wifi; all areas where DR1000 was lacking. Still DR1000 is the best device when it comes to reading and annotating academic papers, which is surprising

given that DR1000 came out 3.5 years ago; perhaps the “eink devices for reading and annotating academic papers” is too niche a niche market to have a successful product. DR1000 was $800 and IREX is now bankrupt.

Anyways, since I am reading novels on Kindle, I have updated my ***old ConTeXt style file*** for DR1000 to also handle Kindle and am releasing that as a ConTeXt module. Actually, as two ConTeXt modules: ***t-eink-devices*** that stores the dimensions and desired font sizes for eink devices (currently, it has data only for DR1000 and Kindle as those are the only devices that I have) and ***t-eink*** that sets an easy to read style that includes:

- Paper size that matches the *screen* dimensions

- Tiny margins, no headers and footers

- Bookmarks for titles and chapters (both DR1000 and Kindle can use PDF bookmarks as table of contents)
- A reasonable default style for chapter and title headings
- A `\startinterlude` … `\stopinterlude` environment for title pages, dedication, etc.

I have only tested this with simple novels (mostly texts and pictures). That is why the module does not set any style for sections, subsections, etc, as I did not need them so far.

This is mostly for personal use, but I am announcing this module in case someone wants to give it a shot. To use the module, simply add

```
\usemodule[irex]
 [
 % alternative=kinde, % or DR1000
```

```
 % mainfont={Tex Gyre Schola},
 % sansfont={Tex Gyre Heros},
 % monofont={Latin Modern Mono},
 % mathfont={Xits},
 % size=, % By default, kindle uses 10pt and DR1000
uses 12pt font.
 % Use this setting if you want to set a font size.
 ]
```

This module passes the font loading to the `simplefonts` module. So, use any name for `mainfont` etc. that `simplefonts` will understand. If you don't set any option, then the default values, indicated above, are used. So, to test out the module, you can just use (for kindle):

```
\usemodule[eink]
```

or (for DR1000)

```
\usemodule[eink][alternative=DR1000]
```

Below are the samples from ***Le Petit Prince***. The text and images were taken from

this website and converted to ConTeXt using ***Pandoc***. The text is also available from ***Project Gutenburg, Australia***.

- ***Kindle sized pdf***
- ***DR1000 sized pdf***

If you have a Kindle or a DR1000, you can compare the quality of these PDFs (hyphenation, line-breaking, widows and orphans) from what you get from the eBook version. If I am to spend 5-10 hours reading a novel, I don't mind spending 15 minutes extra (to create a PDF version of the book) to make that reading experience pleasant.

The output is not perfect, especially in terms for float placement in the Kindle version (Page 5 has an underfull page because the figure was too big to fit in the page, the right float image on page 10 would have been better as

a here figure, the right float figures on page 13-14 are much lower compared to where they are referred, etc.). But, I find these more tolerable than a chapter title appearing at the bottom of the page and occasionally loosing pagination when I highlight text (both of which happen with epub documents).

Reading remote files

Won’t it be nice if TeX could pretty-print files hosted on github, e.g.,

```
\typeRUBYfile{https://raw.github.com/adityam/filter/mast
```

or include a remotely hosted markdown file in your document

```
\processmarkdownfile{https://raw.github.com/adityam/fil
```

I wanted to add this feature to the filter and vim modules.
Although I knew that ConTeXt could ***read remote files directly***, I thought that it would be hard to plug into this mechanism.

Boy, was I wrong. Look at the ***commit history*** of the change needed to add this feature.

All I needed to do, was add `\locfilename` to get the local file name for a file. If the requested file is a remote file (i.e., starts with `http://` or `ftp://`), ConTeXt downloads the file and stores it in the cache directory, and return the name of the cached file. Pretty neat, eh?

With this change, `\process` macro of the filter module can read remote files. Since, the vim module is built on top of the filter module, the `\type` can also read remote files.

The above feature is currently only available in the dev branch. I'll make a new release once I add hooks to force re-download of remote files. Meanwhile, if you have a ConTeXt macro that reads files, just add a `\locfilename` at appropriate place, and your macro will be able to read remote files

Update for the filter module: faster caching

Over the last year, the code base of the ***filter*** module has matured considerably. Now, the module has all the features that I wanted when I started with it about a year and a half back. The last remaining limitation (in my eyes, at least) was that caching of results required a call to external programs (`mtxrun`) to calculate md5 hashes; as such, caching was slow. That is no longer the case. Now (since early December), md5 sums are calculated at the lua end, so there is no time penalty for caching. As a result, in MkIV, recompiling is much faster for

documents having lots of external filter environments with caching enabled(i.e., environments defined with `continue=yes` option).

Since, the ***vim*** module uses the filter module in the background, recompiling MkIV documents using the vim module will also be much faster. In this blog post, I explain both the old and new implementation of caching.

The filter module works as follows. Suppose, you want to use write an

```
\startmarkdown
....
\stopmarkdown
```

to write content in Markdown and use a markdown-to-context tool like pandoc to process the content to ConTeXt. Using the filter module, such an environment can be defined as

```
\defineexternalfilter
   [markdown]
```

```
[filter={pandoc -t context}]
```

This defines a `markdown` start-stop environment. The contents of the environment are written to `\jobname-temp-markdown.tmp` file, which is processed using `pandoc -t context`, the result is written to `\jobname-temp-markdown.tex` file, which is finally read back to ConTeXt.

When a second `markdown` start-stop environment is encountered, the `\jobname-temp-mark` file is overwritten, and the above process is repeated.

The above process works fine for fast programs like `pandoc` but for slow external programs, re-running the external program for each compilation slows down processing time. The filter module allows you to cache results by passing `continue=yes` option:

```
\defineexternalfilter
  [markdown]
  [
    filter={pandoc -t context},
    continue=yes,
  ]
```

Now, the contents of the `markdown` start-stop environment are written to `\jobname-temp-markdown-` file, where is a count for the number of `markdown` environments so far. These files are processed using `pandoc -t context`, and the results are written to `\jobname-temp-markdown-` file, which is finally read back to ConTeXt.

Now, to cache the results, all we need to do is check if the `\jobname-temp-markdown-` file has changed from the previous compilation. If so, re-process the file using `pandoc -t context`; otherwise simply reuse the result of previous compilation.

One low-cost method to check if the con-

tents of a file have changed is to store a md5 sum of the contents and check if the md5 sum of the new file has changed or now. The ConTeXt wrapper script `mtxrun` provides this feature. If you call

```
mtxrun –ifchanged=
```

then `mtxrun` calculates the md5 sum of , stores it in , and runs only if the md5 sum has changed.

I used `mtxrun` with appropriate options to cache the results in the older implementation of the filter module. (This implementation is still in use in MkII). However, this method requires a call to an external program, `mtxrun`, for each filter environment. These external calls to calculate md5 sum might be faster than the call the actual external program (like `pandoc`), but it does take a non-significant amount of time. I had some documents with around 70-80 code

snippets that use the vim module, which in turn uses the filter module; and these 70-80 calls to `mtxrun` took considerable amount of time.

In the new implementation, the md5 sum is computed in Lua and stored in `.tuc` file. No calls to external programs is required; thus, the processing overhead is minimal. In fact, ConTeXt provides a Lua function `job.files.run` that takes care of computing the md5 sum and storing it to the `tuc` file. So, all that I have to do is that instead of calling `mtxrun`, use:

```
\ctxlua{job.files.run("
```

The `job.files.run` function stores the md5 sum in the tuc file, and runs only if the md5 sum has changed. With this implementation, there is very little overhead for multiple md5 sum calculations.

With this change, I think that the filter module is feature complete. From now on, I'll only

be making bugfixes for the filter module and concentrate adding features to my other modules: vim (which is more or less stable now), mathsets (which needs to be re-written for MkIV), and simpleslides (which needs cleanup to keep up with MkIV).

Some thoughts on lowering the learning curve for using TeX (part I)

TeX has a steep learning curve. Often times, steeper than it needs to be. Take, for example, the *special characters* in TeX. Almost every introduction to plain TeX, eplain, LaTeX, or ConTeXt has a section on these *special characters*

```
\ { } $ & # ^ _ & ~
```

A good introduction then goes on to explain why these special characters are important; sometimes dropping a hint about category codes. I feel that these details are useless and, **at the user level**, we should get rid of them.

If you are skeptical, I don't blame you. After all, category codes are the very soul of TeX. However, I strongly believe that they are useless at the user level. Lets go over each of these special characters one-by-one and see if we really need them.

Minimum category codes: \ { }

The only category codes that we need at the user level are \ { }. The character \ marks the start of a control sequence, and { and } group the arguments. The rest, can simply be replaced by control sequences.

Math mode category codes: $ _ ^

In TeX, $ is used to delimit math mode—Knuth used dollars as the math shift character because typesetting math was expensive, so goes an old joke. But do we really need to stick to $? After all, at the user level, both LaTeX and ConTeXt do not use $$ to move to display math

mode. Both macro packages provide environments for display math. Can't we do the same for in-line math? In fact, both LaTeX and ConTeXt also provide macros for in-line math: LaTeX uses `\(...\)` and ConTeXt uses `\m{...}` and `\math{...}`. The only trouble is that these macros are not widely used (and that the LaTeX macros are not robust, but that is easily correctable). The only real argument in favor of `$...$` is that it shorted to type, but compared to `\(...\)` or `\m{...}`, not by much.

The same is true for _ and ^. Both LaTeX and ConTeXt (in fact, so does plain TeX!) provide macros for both of them: `\sp` for ^ and `\sb` for _. But don't panic! I am not asking everyone to start using `\sp` and `\sb`. What I am asking is that _ and ^ have normal meaning in text mode. That is, if I type _, I should get _, not a funky error message. In fact, this is not too difficult to achieve. In LaTeX, use the

underscore package (it is easy to extend that to take care of ^ as well), and in ConTeXt use \nonknuthmode somewhere in your preamble.

Of course, the next logical step is to make $ a normal letter: that is, if you type $ you get $. This is not possible in pdftex, because there is no other means of entering into math mode (other than making some other character the math shift character, but that has the same drawback as making $ the math shift character.) However, luatex provides primitive control sequences for entering and exiting in-line and display math. So, it is possible to make $ a normal letter.

Align character &

Horizontal alignment is one of the strengths of TeX. Most table and multi-line display math environments use horizontal alignment and & specifies the alignment point for horizontal

alignment. Surely, getting rid of & will not work.

Unfortunately, that is true in LaTeX. The & character is so critical for horizontal alignment at the user-level that eliminating it will mean a lot of change. Perhaps, & can be handled in the same manner as _ and ^: it can be a regular letter in text mode and have special meaning inside horizontal alignment. But, it is not always clear to the users which macros use horizontal alignment internally. As such, changing the meaning of & inside some environments will bring more trouble than benefits.

However, the situation in ConTeXt is completely different. At the user-level, & is never used to indicate the alignment point. Both tables and multi-line math display use `\NC... \NC... \NC\NR` type of syntax to indicate new columns. In such a situation it is all the more awkward to explain to a user why & is a special character. It should just be made a normal

letter. LuaTeX provides a `\aligntab` primitive which can be used instead in alignment macros.

Parameter indicator #

Macros is what makes TeX different from all other text markup languages. Automatic numbering, cross-references, headers and footers, and all possible due to macros. And #1 is used to indicate the first parameter for the macro, #2 the second, and so on. But, why do we need this special meaning at the user-level? Only the macro writer needs to care about it.

Most LaTeX macros are written in `.sty` files, that are loaded under a different catcode regime anyways. Most ConTeXt macros are written inside `\unprotect... \protect`. So, it is easy to set the traditional catcode regime in both cases. If a user really needs to define macros in the middle of the document, there can be a “programming” en-

vironment. For example, ConTeXt provides `\starttexcode...\stoptexcode`, which sets the same catcodes as `\unprotect...\protect`. Implementing the same environment in LaTeX is trivial (think `\makeatletter...\makeatother` on steroids).

Unbreakable space ~

Knuth used ~ to indicate an unbreakable space, and that tradition has continued ever since. In this age of Unicode text, do we still need such crutches. It is easy enough to type Unicode `0x00AA` (non-breakable space) in most editors. For example, in vim I just need to type CTRL+K+. A smart syntax highlighting scheme will make the non-breakable space visible. So, there is no real reason to keep on using ~ as a non-breakable space. The same argument holds for the TeX macros for accents, typing in Unicode is easy to input and easy to read (but that will be

the subject of another rant).

So, what's the point of all this?

Now image that all these features have been implemented. Then, we may split the introduction to a TeX macro package into two parts: using the macro package and programming the macro package. Split the first part into two further parts: text mode and math mode. For the text mode, the only special characters are `\ { } %`. All other characters are normal, that means if you type them, you see them in the output (provided the font has the glyph; lets ignore complex languages like Arabic, CJK, and Indic scripts and setting appropriate font features for them at the moment). `\` starts a control sequence, `{...}` groups an argument, and `%` is a line comment. For the math mode, explain how to enter math mode (`\(...\)` or `\m{...}` or the display-math environments) and explain

that _ and ^ are used to indicate sub- and super-scripts. Postpone explaining the programming mode for later. I think that such a scheme will lower the cognitive load on the new user.

Will such a system work? Yes, it will. In fact, it already does. For about an year now, ConTeXt has a `\asciimode` macro that implements all these features, with a slight twist. % is also a normal letter and you need to type to get a line comment (and %{}% if you really need the output). This macro is not enabled by default. I think that making it default will simplify understanding TeX for the first time. As an added advantage, it will also make the job of sanitizing the input simpler for converters (such as ***pandoc***) that convert some other markup language to TeX.

Typewriter scroll mode in vim

I came across a Mac specific application ***Write-room*** for *distraction free writing* and couldn't help wonder that good 'ol ***vim*** provides the same features.

One interesting feature was "typewriter scroll mode", in which, the line that you are writing is always in the middle of the screen. At times, while editing long files, I find myself too often at the bottom of the screen at find myself typing zz to redraw the buffer so that the current line is at the middle of the screen.

It would be nice to have that as default. A quick search of the vim help revealed:

```
'scrolloff' 'so'    number  (default 0)
            global
            {not in Vi}
    Minimal number of screen lines to keep above and
below the cursor.
    This will make some context visible around where
you are working.  If
    you set it to a very large value (999) the cursor line
will always be
    in the middle of the window (except at the start or
end of the file or
    when long lines wrap).
    For scrolling horizontally see 'sidescrolloff'.
    NOTE: This option is set to 0 when 'compatible' is
set.
```

So, the solution is:

```
set scrolloff=99
```

visual counter for Maya numbers

The other day I saw the documentary ***breaking the Maya code***. At one stage they showed how the Maya number system worked. If you haven't seen the documentary, see the wikipedia page on ***Maya number system***. I thought that this would be a nice way to visually represent page numbers in presentations. So, I added a new style to the ***visual counter***, unimaginatively called *mayanumbers*. A ***test*** file comparing this affect with other visual counters from the module is in the ***github*** directory of the module.

Have a look at the output ***pdf***. It uses four counters; cycically from the top-left corner they are: *countdown*, *pulseline*, *scratchcounter*, and finally, *mayanumbers*. !'ll definitely experiment with *mayanumbers* in my next presentation.

Syntax highlighting engines: clean tex output

The vim module uses the vim editor to syntax highlight code snippets in ConTeXt. I thought that it should be straight forward to support other syntax highlighting engines: source-highlight, pygments, HsColor, etc. Unfortunately, that is not the case. None of these syntax highlighting engines were written with reuse in mind.

For example, consider a simple tex file:

```
\definestartstop[important]
                [color=red,
                 style=\italic]
```

```
\starttext
This is an \important{important} text
\stoptext
```

Lets compare the tex file generated by various syntax highlighters:

`source-highlight -f latex` gives

```
% Generator: GNU source-highlight, by Lorenzo Bettini,
http://www.gnu.org/software/src-highlite
\noindent
\mbox{}\textbf{\textcolor{Blue}{\textbackslash{}define
\\

\mbox{}\\\\\\\\\\\\\\\\\[color=red, \\

\mbox{}\\\\\\\\\\\\\\\\\
style=\textbf{\textcolor{Blue}{\textbackslash{}italic}}]
\\

\mbox{}\textbf{\textcolor{Blue}{\textbackslash{}startte
\\

\mbox{}This\ is\ an\
\textbf{\textcolor{Blue}{\textbackslash{}important}}\te
```

```
text \\

\mbox{}\textbf{\textcolor{Blue}{\textbackslash{}stopte
\\

\mbox{}
```

`pygmentize -f latex` gives

```
\begin{Verbatim}[commandchars=\\
\{\}]
\PY{k}{\PYZbs{}definestartstop}\PY{n+na}{[important]
          [color=red,
           style=\PY{k}{\PYZbs{}italic}]
\PY{k}{\PYZbs{}starttext}
This is an
\PY{k}{\PYZbs{}important}\PY{n+nb}{\PYZob{}}impor
text
\PY{k}{\PYZbs{}stoptext}
\end{Verbatim}
```

`HsColor-latex -partial` gives

```
\textcolor{red}{$\backslash$}{\rm{}definestartstop}\te
\hsspace \hsspace \hsspace \hsspace \hsspace
\hsspace \hsspace \hsspace \hsspace \hsspace
\hsspace \hsspace \hsspace \hsspace \hsspace
\hsspace
\textcolor{red}{[}{\rm{}color}\textcolor{red}{=}{\rm{
\hsspace \hsspace \hsspace \hsspace \hsspace
\hsspace \hsspace \hsspace \hsspace \hsspace
\hsspace \hsspace \hsspace \hsspace \hsspace
\hsspace \hsspace
{\rm{}style}\textcolor{cyan}{=$\backslash$}{\rm{}ital
\textcolor{red}{$\backslash$}{\rm{}starttext}\\
{\rm{}This}\hsspace {\rm{}is}\hsspace
{\rm{}an}\hsspace
\textcolor{red}{$\backslash$}{\rm{}important}\textcolo
{\rm{}text}\\
\textcolor{red}{$\backslash$}{\rm{}stoptext}\\
```

HsColor and source-highlight use explicit LaTeX commands for spacing and formatting. Ouch! Pygments uses logical markup, but with cryptic command names. But, from the point of view of using pygments output in ConTeXt, the `\begin{Verbatim}` and `\end{Verbatim}`

are show stopper. (OK, not really. It can be bypassed with some effort).

Based on my experience, I decided to clean up the output generated by `2context.vim`:

```
\SYN[Identifier]{\\definestartstop}[important]
              [color=red,
               style=\SYN[Identifier]{\\italic}]
\SYN[Statement]{\\starttext}
This is an \SYN[Identifier]{\\important}\{important\}
text
\SYN[Statement]{\\stoptext}
```

I assume only four TeX commands to be defined: `\\`, `\{`, and `\}` for backslash, open brace, and close brace; and `\SYN[...]{...}` for syntax highlighting. Thus, if anyone wants to reuse `2context` in plain TeX or LaTeX, or a yet to be written future macro package, they would not need to modify the output at all. I wish the other syntax highlighting programs

did the same.

HTML export

The question of translating TeX to (X)HTML arises frequently. Almost everyone wants it. After all, on the web, (X)HTML is the de-facto standard markup; PDF, with all its hyper link abilities, is clumsier to use. On the other hand, for print, PDF, especially, TeX generated PDF is the de-facto standard (at least for math heavy fields); (X)HTML, with all its print css abilities, is clumsier to use. Often, you want **both** an (X)HTML version and a PDF version of a document. With the popularity of eink devices, epub (which is essentially a zipped (X)HTML file) is also becoming popular. Generating these mul-

tiple *output* formats from the same source is tricky.

The easiest solution, of course, is to use an ascii markup language like markdown, re-structured text, asciidoc, etc. that generate (X)HTML and tex output from the same source. For simple tasks these are great: the synatx is fairly intuitive and there are various tools that translation tools are fairly robust. But, these simple markups lack an important ability of TeX: **programmable macros**. To give an example, suppose I need to write n tuples `$(a_1, \dots,a_n)$` fairly often in a document. When using TeX, I am in the habit of defining a macro

```
\def\TUPLE#1{(#_1,\dots,#_n)}
```

and then use `$\TUPLEa$`. This saves typing and prevents typos. Ascii markup languages lack this ability.

A more robust solution is to use XML as the input language and then use XML-tookchain to translate the input to both HTML and TeX (There are various TeX based solution for parsing XML directly, both in LaTeX and ConTeXt). However, typing XML is a pain, epsecially, when it comes to typing MathML.

A solution which meets the needs midway is inputing text in (X)HTML and math in TeX format. Then use ***mathjax*** to render HTML and one of the TeX XML processing packages to convert to PDF. I think that until browsers become mature in displaying MathML, mathjax is an excellent ad-hoc solution. In addition, it does parse simplist TeX macros. However, you do loose the ability to write complicated TeX macros.

Of course, if you want to parse TeX macros, the only real solution is to use TeX to parse to parse TeX. Why? Because TeX has the *amazing*

ability to change its grammer on the fly.

```
\newcatcodetable \weirdcatcodes
\startcatcodetable \weirdcatcodes
   \catcode`\@ = 0
   \catcode`\( = 1
   \catcode`\) = 2
\stopcatcodetable
\setcatcodetable \weirdcatcodes
```

After this, @ takes the usual meaning of \, (of {, and) of }. So, to start a section, you have to use

```
@section (A section)
```

This ability of changing catcodes is best illustrated by David Carlisle's `xii.tex`

```
\let~\catcode~`76~`A13~`F1~`j00~`P2jdefA71F~`7113
PA''FwPA;;FPAZZFLaLPA//71F71iPAHHFLPAzzFenPASSFthP
A@@FfPARR717273F737271P;ADDFRgniPAWW71FPATTFv
AGGFRruoPAqq71.72.F717271PAYY7172F727171PA??Fi*L
Fjfi71PAVVFjbigskipRPWGAUU71727374
75,76Fjpar71727375Djifx
:76jelse&U76jfiPLAKK7172F71l7271PAXX71FVLnOSeL71S
RrhC?yLRurtKFeLPFovPgaTLtReRomL;PABB71
72,73:Fjif.73.jelse
```

```
B73:jfiXF71PU71
72,73:PWs;AMM71F71diPAJJFRdriPAQQFRsreLPAI
I71Fo71dPA!!FRgiePBt’el@ lTLqdrYmu.Q.,Ke;vz
vzLqpip.Q.,tz;
;Lql.IrsZ.eap,qn.i. i.eLlMaesLdRcna,;!;h
htLqm.MRasZ.ilk,%
s$;z zLqs’.ansZ.Ymi,/sx ;LYegseZRyal,@i;@
TLRlogdLrDsW,@;G
LcYlaDLbJsW,SWXJW ree @rzchLhzsW,;WERcesInW
qt.’oL.Rtrul;e
doTsW,Wk;Rri@stW
aHAHHFndZPpqar.tridgeLinZpe.LtYer.W,:jbye
```

To parse such TeX, you need TeX.

Now, from TeX’s point of view, (X)HTML is not different from any other *backend* like DVI and PDF. It just needs to write the output in a specific format to a file. LuaTeX makes this easy and ConTeXt MkIV now supports a XHTML backend. Simply add

```
\setupbackend[export=yes, xhtml=yes]
```

in your preamble. This creates a

\joname.xhtml file (and a \jobname.export XML file) For example

```
\setupbackend[export=yes, xhtml=yes]

\starttext
\input ward
\stoptext
```

gives

This is not really an XHTML file. It is just an XML file. With a proper stylesheet, most browsers will be able to display this file. Creating such a stylesheet is easy. A simple example of such a stylesheet is ***here***. To use this stylesheet, save it as, say mkiv-export.css, then add

```
\setupbackend[export=yes, xhtml=yes,
css=mkiv-export.css]
```

This adds the following line to the generated XHTML output

The XHTML file exports the **structure** of the document, not the style. Consider the following, more complicated example:

```
\setupbackend[export=yes, xhtml=yes,
css=mkiv-export.css]

\definestartstop[important][style=italic]

\starttext
\section {The first attempt}

The XHTML export with \important{structured text},
but not manual {\em style}
\italic{commands}. But the good thing is that it works
with math!

Consider a quadratic equation $ax^2 + bx + c = 0$.
The roots of this equation are
\startformula
  x = (-b ± \sqrt{b^2 - 4ac})/2a
\stopformula
\stoptext
```

This gives

Notice some features of the output: the section number is exported with the section title (if you change the conversion of the section number, the output will honor that); the structure command `\important` is exported, the style commands `\em` and `\italic` are not; math is exported as MathML with Unicode symbols!. There are some interesting features that we are experimenting with. I'll post more about the MathML export in the future.

So, if you are interested in XHTML output for TeX sources, play around with the new export feature. It is not perfect … yet. But with MathML and SVG (remeber, Metapost has a SVG backend), it is possible to get a fully working XHTML output for the TeX input *generated by TeX* rather than a pre- or post-processor. After all, **only TeX can parse TeX**

```
\setupbackend[export=yes, xhtml=yes,
css=mkiv-export.css]
```

```
\starttext
\let\bye=\donothing %xii.tex ends with \bye
\input xii
\stoptext
```

(The output is ***here***)

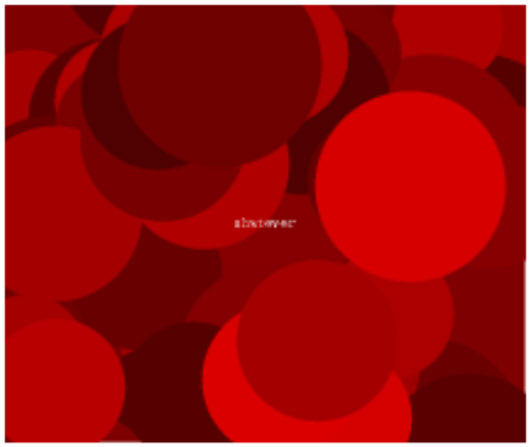
Images for documentation examples

ConTeXt users tend to use the *famous* Dutch cow as a place holder image for documentation examples. At times, it gets ***annoying***. One alternative is to use random figures:

```
\useMPlibrary[dum]

\starttext
  \placefigure[left,none]{}
    {\externalfigure[whatever][width=0.5\textwidth,
height=0.3\textheight]}
  \input knuth
\stoptext
```

which gives an image as shown below:



Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large–scale user; the designer should also write the first user manual.
The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.
But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

The image is draws random circles with random colors using metapost.

In a recent ***discussion*** on ***tex.se***, Sharpie mentioned a website: ***http://placekitten.com***, that provides pictures of kittens as placeholders for images. Those kittens are definitely cuter than the Dutch cow. Using these kittens as placeholder images is easy: if you want a 300pt x 400pt images, just type:

```
\externalfigure[http://placekitten.com/g/300/400][method
```

ConTeXt takes care of downloading and

caching the image. However, to use this approach, you need to know the required image dimensions in postscript points. Many a times, the image dimensions are known in relative terms like the above example where we asked for an image that is `0.5\textwidth` wide and `0.3\textheight` high. To convert a arbitrary TeX dimention to points, we can use `\dimexpr` which reports its result in points. So, to know how much is `0.5\textwidth` we can use

```
\the\dimexpr0.5\textwidth
```

For the default setup, this reports `213.3937pt`. To use this dimension with place kitten website, we need to get rid of the `pt` suffix—a freqeuntly needed feature for which ConTeXt, like almost all other macro packages, provides a built-in macro: `\withoutpt`. Thus, we can use

```
\withoutpt{\the\dimexpr0.5\textwidth}
```

to get the value of `0.5\textwidth` without the `pt` suffix. However, this is not enough: the place kitten website does not provide images with fractional sizes. So, we need to round (or truncate) the value to a whole number. I don't know if ConTeXt has a built-in macro for that; neither do I care enough to write such a macro in TeX. Truncating a float to an integer is easy in Lua, and that is what I will use. Combining all this, here is a macro that places an image of a kitten that is of a particular size:

```
\def\externalkitten[#1]%
   {\getparameters[kitten][width=10pt, height=10pt,
#1]
    \externalfigure

[\ctxlua{context("http://placekitten.com/g/\%0.0f/\%0.0f",
              \withoutpt{\the\dimexpr\kittenwidth},
              \withoutpt{\the\dimexpr\kittenheight})}]
      [#1, method=jpg]}
```

This can be used as:

```
\starttext
  \placefigure[left,none]{}
    {\externalkitten[width=0.5\textwidth,
height=0.3\textheight]}
  \input knuth
\stoptext
```

which gives the following result:

Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large–scale user; the designer should also write the first user manual.

The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.

But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

I think that this looks much nicer than the random image above. Below is an example that uses the place kitten images to show how the `location` key works.

```
\startcombination[3*3]
  {\externalkitten[width=0.2\textwidth,
```

```
height=0.15\textheight, location=top]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.20\textheight, location=top]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.25\textheight, location=top]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.15\textheight, location=middle]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.20\textheight, location=middle]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.25\textheight, location=middle]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.15\textheight, location=bottom]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.20\textheight, location=bottom]}{}
  {\externalkitten[width=0.2\textwidth,
height=0.25\textheight, location=bottom]}{}
\stopcombination
```

Note that all three images are of different sizes. The first row is top aligned, the second row is middle aligned, and the third row is bottom aligned. Now imagine how this would have looked if all three images were scaled cows!

Adding color to tables

Texblog had an interesting post on creating tables with alternating colors. See the ***pdf*** for the final output. I thought that it will be interesting to see how to reproduce the same effect in ConTeXt. This is the table that I am going to use for my tests:

```
\startbuffer[contents]
\NC      \NC Col 1 \NC Col 2 \NC Col 3 \NC Col 4 \NC Col
5 \NC \NR
\NC Row 1 \NC 1    \NC 2    \NC 3    \NC 4    \NC 5
\NC \NR
\NC Row 2 \NC 1    \NC 2    \NC 3    \NC 4    \NC 5
\NC \NR
\NC Row 3 \NC 1    \NC 2    \NC 3    \NC 4    \NC 5
\NC \NR
```

```
\NC Row 4 \NC 1    \NC 2    \NC 3    \NC 4    \NC 5
\NC \NR
\NC Row 5 \NC 1    \NC 2    \NC 3    \NC 4    \NC 5
\NC \NR
\NC Row 6 \NC 1    \NC 2    \NC 3    \NC 4    \NC 5
\NC \NR
\stopbuffer
```

First lets get the frames right. The example typesets each table with a rules around the first row, a vertical rule after the first column, and a rule after the last column. To get that, use

```
\startsetups table:frame
\setupTABLE[each][each][frame=off, align=middle,
background=color, rulethickness=1bp]
\setupTABLE[row][first][bottomframe=on,topframe=on]
\setupTABLE[row][last] [bottomframe=on]
\setupTABLE[column][first][rightframe=on]
\stopsetups
```

That is pretty self explanatory. By default, all cells in a table have a frame, so I switch that off with `frame=off`; each cell should be middle aligned, so I use `align=middle`;

`background=color` is for later when I will set the color for the table; `rulethickness` sets the thickness of the line. This setup gives the following result:

```
\startTABLE[setups=table:frame]
  \getbuffer[contents]
\stopTABLE
```

	Col 1	Col 2	Col 3	Col 4	Col 5
Row 1	1	2	3	4	5
Row 2	1	2	3	4	5
Row 3	1	2	3	4	5
Row 4	1	2	3	4	5
Row 5	1	2	3	4	5
Row 6	1	2	3	4	5

Now, lets try the different color setups.

Rows with alternative colors

This one is simple. Set

```
\startsetups table:rows
\setupTABLE[row][odd] [backgroundcolor=yellow:1]
\setupTABLE[row][even][backgroundcolor=blue:1]
```

```
\stopsetups
```

The colors `yellow:1` and `blue:1` mean 10%yellow and blue. Then add this setup to the table to get the following result:

```
\startTABLE[setups={table:frame,table:rows}]
  \getbuffer[contents]
\stopTABLE
```

	Col 1	Col 2	Col 3	Col 4	Col 5
Row 1	1	2	3	4	5
Row 2	1	2	3	4	5
Row 3	1	2	3	4	5
Row 4	1	2	3	4	5
Row 5	1	2	3	4	5
Row 6	1	2	3	4	5

Columns with alternative colors

This is almost identical to the row setup.

```
\startsetups table:columns
\setupTABLE[column][odd]
[backgroundcolor=yellow:1]
\setupTABLE[column][even][backgroundcolor=blue:1]
```

```
\stopsetups

\startTABLE[setups={table:frame,table:columns}]
  \getbuffer[contents]
\stopTABLE
```

	Col 1	Col 2	Col 3	Col 4	Col 5
Row 1	1	2	3	4	5
Row 2	1	2	3	4	5
Row 3	1	2	3	4	5
Row 4	1	2	3	4	5
Row 5	1	2	3	4	5
Row 6	1	2	3	4	5

Chessboard coloring

And now for the most interesting setup: coloring the table cells like a chessboard.

```
\startsetups table:chessboard
\setupTABLE[each][each][backgroundcolor=yellow:1]
\setupTABLE[odd][odd]  [backgroundcolor=blue:1]
\setupTABLE[even][even][backgroundcolor=blue:1]
\stopsetups

\startTABLE[setups={table:frame,table:chessboard}]
```

```
  \getbuffer[contents]
\stopTABLE
```

	Col 1	Col 2	Col 3	Col 4	Col 5
Row 1	1	2	3	4	5
Row 2	1	2	3	4	5
Row 3	1	2	3	4	5
Row 4	1	2	3	4	5
Row 5	1	2	3	4	5
Row 6	1	2	3	4	5