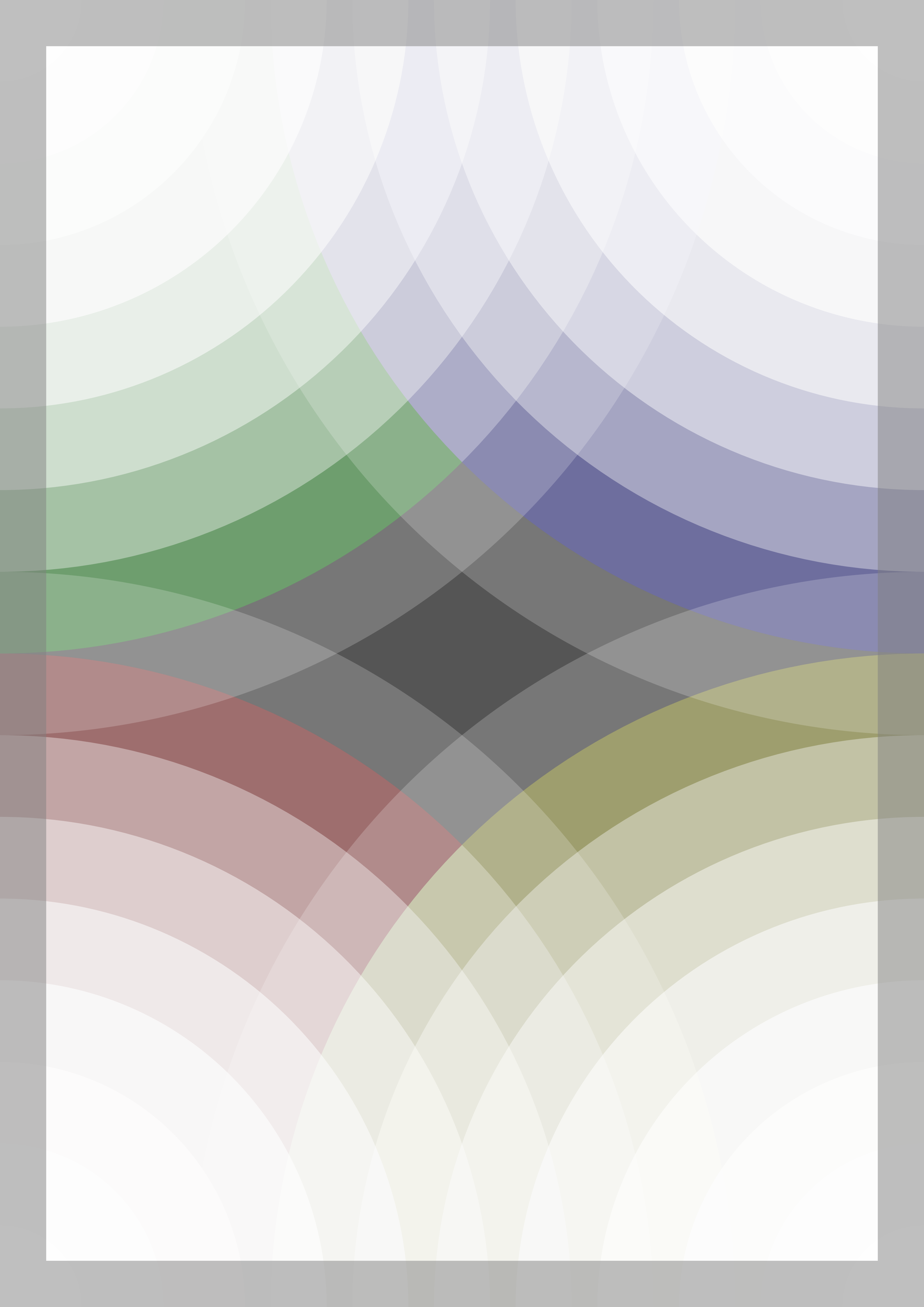


SciTE version 3.61 works ok but 3.62 crashes. It'a a real pity that SciTE doesn't have the scintillua lexer built in, which would also make integration a bit nicer by sharing the Lua instance. The ConTeXt lexing discussed here is the lexing I assume when using ConTeXt MkIV, but alas it's not easy to get it running on Unix and on MacOSX there is no Lua lexing available.

For a long time at Pragma ADE we used TeXedit, an editor we'd written in Modula. It had some project management features and recognized the project structure in ConTeXt documents. Later we rewrote this to a platform independent reimplementation called TeXwork written in Perl/Tk (not to be confused with the editor with the plural name).

In the beginning of the century I can into SciTE, written by Neil Hodgson. Although the mentioned editors provide some functionality not present in SciTE we decided to use that editor because it frees us from maintaining our own. I ported our TeX and MetaPost (line based) syntax highlighting to SciTE and got a lot of others for free.

After a while I found out that there was an extension interface written in Lua. I played with it and wrote a few extensions too. This pleasant experience later triggered the LuaTeX project.

A decade into the century SciTE got another new feature: you can write dynamic external lexers in Lua using lpeg. As in the meantime ConTeXt has evolved in a TeX/Lua hybrid, it made sense to look into this. The result is a couple of lexers that suit TeX, MetaPost and Lua usage in ConTeXt MkIV. As we also use xml as input and output format a lexer for xml is also provided. And because pdf is one of the backend formats lexing of pdf is also implemented.[1]

In the ConTeXt (standalone) distribution you will find the relevant files under:

```
<texroot>/tex/texmf-context/context/data/scite
```

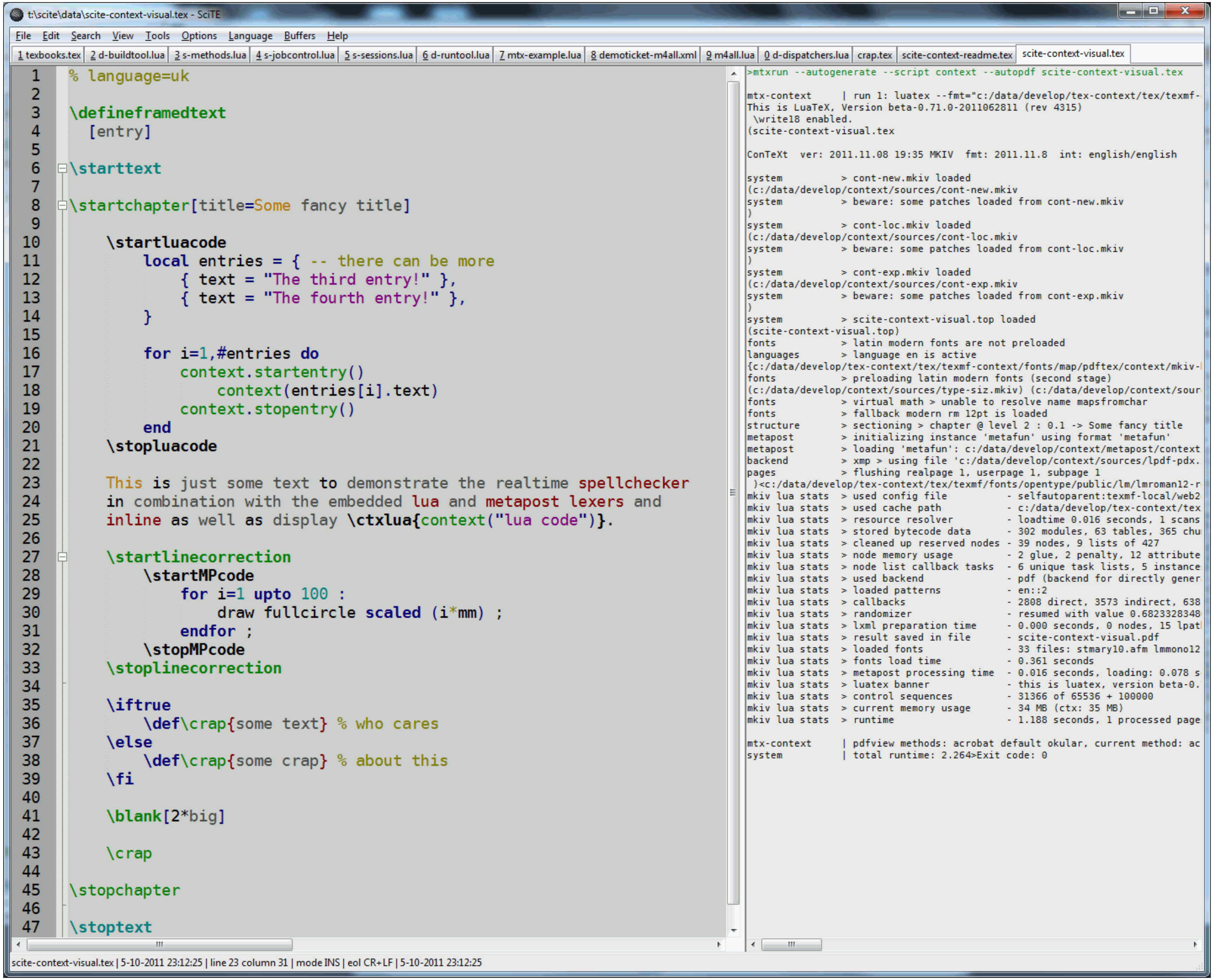
Normally a user will not have to dive into the implementation details but in principle you can tweak the properties files to suit your purpose.

The color scheme that we use is consistent over the lexers but we use more colors that in the traditional lexing. For instance, TeX primitives, low level TeX commands, TeX constants, basic file structure related commands, and user commands all get a different treatment. When spell checking is turned on, we indicate unknown words, but also words that are known but might need checking, for instance because they have an uppercase character. In figure 1 we some of that in practice.

Installing SciTE is straightforward. We are most familiar with MS Windows but for other operating systems installation is not much different. First you need to fetch the archive from:

---

[1] In the process some of the general lexing framework was adapted to suit our demands for speed. We ship these files as well.



```
% language=uk

\defineframedtext
  [entry]

\starttext

\startchapter[title=Some fancy title]

    \startluacode
        local entries = { -- there can be more
            { text = "The third entry!" },
            { text = "The fourth entry!" },
        }

        for i=1,#entries do
            context.startentry()
                context(entries[i].text)
            context.stopentry()
        end
    \stopluacode

    This is just some text to demonstrate the realtime spellchecker
    in combination with the embedded lua and metapost lexers and
    inline as well as display \ctxlua{context("lua code")}.

    \startlinecorrection
        \startMPcode
            for i=1 upto 100 :
                draw fullcircle scaled (i*mm) ;
            endfor ;
        \stopMPcode
    \stoplinecorrection

    \iftrue
        \def\crap{some text} % who cares
    \else
        \def\crap{some crap} % about this
    \fi

    \blank[2*big]

    \crap

\stopchapter

\stoptext
```

```
>mtxrun --autogenerate --script context --autopdf scite-context-visual.tex

mtx-context     | run 1: luatex --fmt="c:/data/develop/tex-context/tex/texmf-
This is LuaTeX, Version beta-0.71.0-2011062811 (rev 4315)
 \write18 enabled.
(scite-context-visual.tex

ConTeXt  ver: 2011.11.08 19:35 MKIV  fmt: 2011.11.8  int: english/english

system          > cont-new.mkiv loaded
(c:/data/develop/context/sources/cont-new.mkiv
system          > beware: some patches loaded from cont-new.mkiv
)
system          > cont-loc.mkiv loaded
(c:/data/develop/context/sources/cont-loc.mkiv
system          > beware: some patches loaded from cont-loc.mkiv
)
system          > cont-exp.mkiv loaded
(c:/data/develop/context/sources/cont-exp.mkiv
system          > beware: some patches loaded from cont-exp.mkiv
)
system          > scite-context-visual.top loaded
(scite-context-visual.top)
fonts           > latin modern fonts are not preloaded
languages       > language en is active
{c:/data/develop/tex-context/tex/texmf-context/fonts/map/pdftex/context/mkiv-
fonts           > preloading latin modern fonts (second stage)
(c:/data/develop/context/sources/type-siz.mkiv) (c:/data/develop/context/sour
fonts           > virtual math > unable to resolve name mapsfromchar
fonts           > fallback modern rm 12pt is loaded
structure       > sectioning > chapter @ level 2 : 0.1 -> Some fancy title
metapost        > initializing instance 'metafun' using format 'metafun'
metapost        > loading 'metafun': c:/data/develop/context/metapost/context
backend         > xmp > using file 'c:/data/develop/context/sources/lpdf-pdx.
pages           > flushing realpage 1, userpage 1, subpage 1
 )<c:/data/develop/tex-context/tex/texmf/fonts/opentype/public/lm/lmroman12-r
mkiv lua stats  > used config file          - selfautoparent:texmf-local/web2
mkiv lua stats  > used cache path           - c:/data/develop/tex-context/tex
mkiv lua stats  > resource resolver         - loadtime 0.016 seconds, 1 scans
mkiv lua stats  > stored bytecode data      - 302 modules, 63 tables, 365 chu
mkiv lua stats  > cleaned up reserved nodes - 39 nodes, 9 lists of 427
mkiv lua stats  > node memory usage         - 2 glue, 2 penalty, 12 attribute
mkiv lua stats  > node list callback tasks  - 6 unique task lists, 5 instance
mkiv lua stats  > used backend              - pdf (backend for directly gener
mkiv lua stats  > loaded patterns           - en::2
mkiv lua stats  > callbacks                 - 2808 direct, 3573 indirect, 638
mkiv lua stats  > randomizer                - resumed with value 0.6823328348
mkiv lua stats  > lxml preparation time     - 0.000 seconds, 0 nodes, 15 lpat
mkiv lua stats  > result saved in file      - scite-context-visual.pdf
mkiv lua stats  > loaded fonts              - 33 files: stmary10.afm lmmono12
mkiv lua stats  > fonts load time           - 0.361 seconds
mkiv lua stats  > metapost processing time  - 0.016 seconds, loading: 0.078 s
mkiv lua stats  > luatex banner             - this is luatex, version beta-0.
mkiv lua stats  > control sequences         - 31366 of 65536 + 100000
mkiv lua stats  > current memory usage      - 34 MB (ctx: 35 MB)
mkiv lua stats  > runtime                   - 1.188 seconds, 1 processed page

mtx-context     | pdfview methods: acrobat default okular, current method: ac
system          | total runtime: 2.264>Exit code: 0
```

Nested lexers in action.

```
www.scintilla.org
```

The MS Windows binaries are zipped in `wscite.zip`, and you can unzip this in any directory you want as long as you make sure that the binary ends up in your path or as shortcut on your desktop. So, say that you install SciTE in:

```
c:\data\system\scite\wscite
```

You need to add this path to your local path definition. Installing SciTE to some known place has the advantage that you can move it around. There are no special dependencies on the operating system.

On MS Windows you can for instance install SciTE in:

```
c:\data\system\scite
```

and then end up with:

```
c:\data\system\scite\wscite
```

and that is the path you need to add to your environment `PATH` variable.

On linux the files end up in:

```
/usr/bin
/usr/share/scite
```

Where the second path is the path we will put more files.

scintillua

Next you need to install the lpeg lexers.[2] The library is part of the `textadept` editor by Mitchell (mitchell .att.foicica.com) which is also based on scintilla: The archive can be fetched from:

```
http://foicica.com/scintillua/
```

On MS Windows you need to copy the files to the `wscite` folder (so we end up with a `lexers` subfolder there). For linux the place depends on the distribution, for instance `/usr/share/scite`; this is the place where the regular properties files live.[3]

So, you end up, on MS Windows with:

```
c:\data\system\scite\wscite\lexers
```

And on linux:

```
/usr/share/scite/lexers
```

Beware: if you're on a 64 bit system, you need to rename the 64 bit `so` library into one without a number. Unfortunately the 64 bit library is now always available which can give surprises when the operating system gets updates. In such a case you should downgrade or use `wine` with the MS Windows

---

[2] Versions later than 2.11 will not run on MS Windows 2K. In that case you need to comment the external lexer import.

[3] If you update, don't do so without testing first. Sometimes there are changes in SciTE that influence the lexers in which case you have to wait till we have update them to suit those changes.

binaries instead. After installation you need to restart SciTE in order to see if things work out as expected.

When we started using this nice extension, we ran into issues and as a consequence shipped a patched Lua code. We also needed some more control as we wanted to provide more features and complex nested lexers. Because the library api changed a couple of times, we now have our own variant which will be cleaned up over time to be more consistent with our other Lua code (so that we can also use it in ConTEXt as variant verbatim lexer). We hope to be able to use the `scintillua` library as it does the job.

Anyway, if you want to use ConTEXt, you need to copy the relevant files from

```
<texroot>/tex/texmf-context/context/data/scite
```

to the path were SciTE keeps its property files (`*.properties`). This is the path we already mentioned. There should be a file there called `SciteGlobal.properties`.

So,in the end you get on MS Windows new files in:

```
c:\data\system\scite\wscite
c:\data\system\scite\wscite\context
c:\data\system\scite\wscite\context\lexer
c:\data\system\scite\wscite\context\lexer\themes
c:\data\system\scite\wscite\context\lexer\data
c:\data\system\scite\wscite\context\documents
```

while on linux you get:

```
/usr/bin/share/
/usr/bin/share/context
/usr/bin/share/context/lexer
/usr/bin/share/context/lexer/themes
/usr/bin/share/context/lexer/data
/usr/bin/share/context/documents
```

At the end of the `SciteGlobal.properties` you need to add the following line:

```
import context/scite-context-user
```

After this, things should run as expected (given that TEX runs at the console as well).

The configuration file defaults to the Dejavu fonts. These free fonts are part of the ConTEXt suite (also known as the standalone distribution). Of course you can fetch them from `http://dejavu-fonts.org` as well. You have to copy them to where your operating system expects them. In the suite they are available in:

```
<contextroot>/tex/texmf/fonts/truetype/public/dejavu
```

Just a quick note to some extensions. If you select a part of the text (normally you do this with the shift key pressed) and you hit `Shift-F11`, you get a menu with some options. More (robust) ones will be provided at some point.

If you want to have spell checking, you need have files with correct words on each line. The first line of a file determines the language:

```
% language=uk
```

When you use the external lexers, you need to provide some files. Given that you have a text file with valid words only, you can run the following script:

```
mtxrun --script scite --words nl uk
```

This will convert files with names like `spell-nl.txt` into Lua files that you need to copy to the `lexers/data` path. Spell checking happens realtime when you have the language directive (just add a bogus character to disable it). Wrong words are colored red, and words that might have a case problem are colored orange. Recognized words are greyed and words with less than three characters are ignored.

A spell checking file has to be put in the `lexers/data` directory and looks as follows (e.g. `spell-uk.lua`):

```
return {
 ["max"]=40,
 ["min"]=3,
 ["n"]=151493,
 ["words"]={
  ["aardvark"]="aardvark",
  ["aardvarks"]="aardvarks",
  ["aardwolf"]="aardwolf",
  ["aardwolves"]="aardwolves",
  ...
 }
}
```

The keys are words that get checked for the given value (which can have uppercase characters). The word files are not distributed (but they might be at some point).

In the case of internal lexers, the following file is needed:

```
spell-uk.txt
```

If you use the traditional lexer, this file is taken from the path determined by the environment variable:

```
CTXSPELLPATH
```

As already mentioned, the lpeg lexer expects them in the data path. This is because the Lua instance that does the lexing is rather minimalistic and lacks some libraries as well as cannot access the main SciTE state.

Spell checking in `txt` files is enabled by adding a first line:

```
[#!-%] language=uk
```

The first character on that line is one of the four mentioned between square brackets. So,

```
# language=uk
```

should work. For xml files there are two methods. You can use the following (at the start of the file):

```
<?xml ... language="uk" ?>
```

But probably better is to use the next directive just below the usual xml marker line:

```
<?context-directive editor language uk ?>
```

In a similar fashion you can drive the interface checking:

```
% interface=nl
```

The internal lexers are controlled by the property files while the external ones are steered with themes. Unfortunately there is hardly any access to properties from the external lexer code nor can we consult the file system and/or run programs like `mtxrun`. This means that we cannot use configuration files in the ConTEXt distribution directly. Hopefully this changes with future releases.

These are the more advanced lexers. They provide more detail and the ConTEXt lexer also supports nested MetaPost and Lua. Currently there is no detailed configuration but this might change once they are stable.

The external lexers operate on documents while the internal ones operate on lines. This can make the external lexers slow on large documents. We've optimized the code somewhat for speed and memory consumption but there's only so much one can do. While lexing each change in style needs a small table but allocating and garbage collecting many small tables comes at a price. Of course in practice this probably gets unnoticed.[4]

The external lpeg lexers work okay with the MS Windows and linux versions of SciTE, but unfortunately at the time of writing this, the Lua library that is needed is not available for the MacOSX version of SciTE. Also, due to the fact that the lexing framework is rather isolated, there are some issues that cannot be addressed in the properly, at least not currently.

In addition to ConTEXt and MetaFun lexing a Lua lexer is also provided so that we can handle ConTEXt Lua Document (cld) files too. There is also an xml lexer. This one also provides spell checking. The

---

[4] I wrote the code in 2011 on a more than 5 years old Dell M90 laptop, so I suppose that speed is less an issue now.

pdf lexer tries to do a good job on pdf files, but it has some limitations. There is also a simple text file lexer that does spell checking. Finally there is a lexer for cweb files.

Don't worry if you see an orange rectangle in your TeX or xml document. This indicates that there is a special space character there, for instance `0xA0`, the nonbreakable space. Of course we assume that you use utf8 as input encoding.

SciTE has quite some built in lexers. A lexer is responsible for highlighting the syntax of your document. The way a TeX file is treated is configured in the file:

```
tex.properties
```

You can edit this file to your needs using the menu entry under `options` in the top bar. In this file, the following settings apply to the TeX lexer:

```
lexer.tex.interface.default=0
lexer.tex.use.keywords=1
lexer.tex.comment.process=0
lexer.tex.auto.if=1
```

The option `lexer.tex.interface.default` determines the way keywords are highlighted. You can control the interface from your document as well, which makes more sense that editing the configuration file each time.

```
% interface=all|tex|nl|en|de|cz|it|ro|latex
```

The values in the properties file and the keywords in the preamble line have the following meaning:

0	`all`	all commands (preceded by a backslash)
1	`tex`	TeX, ε-TeX, pdfTeX, Omega primitives (and macros)
2	`nl`	the dutch ConTeXt interface
3	`en`	the english ConTeXt interface
4	`de`	the german ConTeXt interface
5	`cz`	the czech ConTeXt interface
6	`it`	the italian ConTeXt interface
7	`ro`	the romanian ConTeXt interface
8	`latex`	LaTeX (apart from packages)

The configuration file is set up in such a way that you can easily add more keywords to the lists. The keywords for the second and higher interfaces are defined in their own properties files. If you're curious about the way this is configures, you can peek into the property files that start with `scite-context`. When you have ConTeXt installed you can generate configuration files with

```
mtxrun --script interface --scite
```

You need to make sure that you move the result to the right place so best not mess around with this command and use the files from the distribution.

Back to the properties in `tex.properties`. You can disable keyword coloring alltogether with:

```
lexer.tex.use.keywords=0
```

but this is only handy for testing purposes. More interesting is that you can influence the way comment is treated:

```
lexer.tex.comment.process=0
```

When set to zero, comment is not interpreted as TeX code and it will come out in a uniform color. But, when set to one, you will get as much colors as a TeX source. It's a matter of taste what you choose.

The lexer tries to cope with the TeX syntax as good as possible and takes for instance care of the funny `^^` notation. A special treatment is applied to so called `\if`'s:

```
lexer.tex.auto.if=1
```

This is the default setting. When set to one, all `\ifwhatever`'s will be seen as a command. When set to zero, only the primitive `\if`'s will be treated. In order not to confuse you, when this property is set to one, the lexer will not color an `\ifwhatever` that follows an `\newif`.

The MetaPost lexer is set up slightly different from its TeX counterpart, first of all because MetaPost is more a language that TeX. As with the TeX lexer, we can control the interpretation of identifiers. The MetaPost specific configuration file is:

```
metapost.properties
```

Here you can find properties like:

```
lexer.metapost.interface.default=1
```

Instead of editing the configuration file you can control the lexer with the first line in your document:

```
% interface=none|metapost|mp|metafun
```

The numbers and keywords have the following meaning:

`0`	`none`	no highlighting of identifiers
`1`	`metapost or mp`	MetaPost primitives and macros
`2`	`metafun`	MetaFun macros

Similar to the TeX lexer, you can influence the way comments are handled:

```
lexer.metapost.comment.process=1
```

This will interpret comment as MetaPost code, which is not that useful (opposite to TeX, where documentation is often coded in TeX).

The lexer will color the MetaPost keywords, and, when enabled also additional keywords (like those of MetaFun). The additional keywords are colored and shown in a slanted font.

The MetaFun keywords are defined in a separate file:

```
metafun-scite.properties
```

You can either copy this file to the path where you global properties files lives, or put a copy in the path of your user properties file. In that case you need to add an entry to the file `SciTEUser.properties`:

```
import metafun-scite
```

The lexer is able to recognize `btex`–`etex` and will treat anything in between as just text. The same happens with strings (between `"`). Both act on a per line basis.

When `mtxrun` is in your path, ConTEXt should run out of the box. You can find `mtxrun` in:

```
<contextroot>/tex/texmf-mswin/bin
```

or in a similar path that suits the operating system that you use.

When you hit `CTRL-12` your document will be processed. Take a look at the `Tools` menu to see what more is provided.

When the Lua extensions are loaded, you will see a message in the log pane that looks like:

```
-  see scite-ctx.properties for configuring info

-  ctx.spellcheck.wordpath set to ENV(CTXSPELLPATH)
-  ctxspellpath set to c:\data\develop\context\spell
-  ctx.spellcheck.wordpath expands to c:\data\develop\context\spell

-  ctx.wraptext.length is set to 65
-  key bindings:

Shift + F11   pop up menu with ctx options

Ctrl  + B     check spelling
Ctrl  + M     wrap text (auto indent)
Ctrl  + R     reset spelling results
Ctrl  + I     insert template
Ctrl  + E     open log file
Ctrl  + +     show language character strip (key might change)

-  recognized first lines:

xml   <?xml version='1.0' language='nl'
tex   % language=nl
```

This message tells you what extras are available. The language character strip feature is relatively new and displays buttons at the bottom of the screen for the characters in a (chosen) language. This is handy when you occasionally have to key in (snippets) of a language you're not familiar with. More alphabets will be added (we take data from some ConTEXt language relates files).

It is possible to define (and use) templates. There is a demo file in the distribution called `scite-ctx-templates.` You can put a similar file in your working path or one or two levels up from there. If not found, the default (demo) file will be used. a manu is called up with `ctrl-i`.

A template file is a Lua file and looks like this:

```
return {
    xml = {
        {
            name     = "bold",
            nature   = "inline",
            template = "<b>?</b>",
        },
        {
            name     = "p",
            nature   = "display",
            template = "<p>?</p>",
        },
        {
            name     = "emphasized",
            nature   = "inline",
            template = "<em>?</em>",
        },
        {
            name     = "inline",
            nature   = "inline",
            template = "<m>?</m>",
        },
        {
            name     = "display",
            nature   = "display",
            template = "<math>?</math>",
        },
        {
            name     = "itemize",
            nature   = "display",
            template =
[[<itemize>
    <item>?</item>
    <item>?</item>
    <item>?</item>
</itemize>]],
        },
    },
}
```

In xml sources you can add a line:

```
<?context-directive job ctxtemplate mytemplates.lua ?>
```

The file will be searched for in the current direct and upto two levels higher. When no file is found the TEX distribution is checked.

The files `scite-ctx-example` and `scite-ctx-context` define the menu commands, like:

```
command.25.$(file.patterns.example)=insert_template
```

The following keybindings are available in SciTE. Most of this list is taken from the on-line help pages.

Ctrl+Keypad+	magnify text size
Ctrl+Keypad-	reduce text size
Ctrl+Keypad/	restore text size to normal
Ctrl+Keypad*	expand or contract a fold point
Ctrl+Tab	cycle through recent files
Tab	indent block
Shift+Tab	dedent block
Ctrl+BackSpace	delete to start of word
Ctrl+Delete	delete to end of word
Ctrl+Shift+BackSpace	delete to start of line
Ctrl+Shift+Delete	delete to end of line
Ctrl+Home	go to start of document; Shift extends selection
Ctrl+End	go to end of document; Shift extends selection
Alt+Home	go to start of display line; Shift extends selection
Alt+End	go to end of display line; Shift extends selection
Ctrl+F2	create or delete a bookmark
F2	go to next bookmark
Ctrl+F3	find selection
Ctrl+Shift+F3	find selection backwards
Ctrl+Up	scroll up
Ctrl+Down	scroll down
Ctrl+C	copy selection to buffer
Ctrl+V	insert content of buffer
Ctrl+X	copy selection to buffer and delete selection
Ctrl+L	line cut
Ctrl+Shift+T	line copy
Ctrl+Shift+L	line delete
Ctrl+T	line transpose with previous
Ctrl+D	line duplicate
Ctrl+K	find matching preprocessor conditional, skipping nested ones
Ctrl+Shift+K	select to matching preprocessor conditional
Ctrl+J	find matching preprocessor conditional backwards, skipping nested ones
Ctrl+Shift+J	select to matching preprocessor conditional backwards
Ctrl+[	previous paragraph; Shift extends selection
Ctrl+]	next paragraph; Shift extends selection
Ctrl+Left	previous word; Shift extends selection
Ctrl+Right	next word; Shift extends selection
Ctrl+/	previous word part; Shift extends selection
Ctrl+\	next word part; Shift extends selection
F12 / Ctrl+F7	check (or process)

`Ctrl+F12 / Ctrl+F7`	process (run)
`Alt+F12 / Ctrl+F7`	process (run) using the luajit vm (if applicable)

author	Hans Hagen
copyright	PRAGMA ADE, Hasselt NL
more info	www.pragma-ade.com www.contextgarden.net
version	October 8, 2018