AGDoc Formatter

Yet Another Document Formatting Tool

Introduction

AGDoc is a command line utility that compiles text files and produces documents. This is rather an indistinct definition, but it means the following.

- Simple and intuitive source syntax.
- Multiple output formats.
- Configurable markup elements.
- Automatic collecting and formatting of tables of contents.
- Collecting of the index and automatic cross referencing within the project.
- Automatic syntax highlighting of your source code.
- Easy adaptation for 16- and 32-bit Unicode.

It also means that you will need to spend some time configuring the output formats and creating your own templates, but then you can put all your efforts into writing the text.

Generally speaking, it all looks like **TEX**, but in practice it is much simpler in use, much more compact, and of course, much less automated. Compared to **TEX** it's just a tiny ant crowling beside an elephant. For example, the full installation of **MiKTEX** weighs about 250 megabytes, while **AGDoc** is only 310 kilobytes of source code in **C++**.

Rationale

The main purpose of **AGDoc** is to provide a simple document formatting tool for developers of Open

Source software. **AGDoc** probably doesn't fit the needs of writing books with many formulae, but it perfectly solves the task of formatting compact reference manuals for your projects. The following is a brief overview of the formatting systems that were considered before I decided to create **AGDoc**.

NOTE
I do not want to say they are bad tools. They all are great. I just want to rationalize my personal aspirations when creating the **AGDoc** formatter.

- **Microsoft Word**. It's a nice program to compose business letters, but it's hard to use it when writing large amount of text. Below is just a little aspect. You can define auto replacements with custom formatting, for example, you type "TEX" and it's being replaced to **TEX** with a hyperlink. But the problem is the rules of replacement don't exist in the source text, it only works when you **type** the text. If you decide to change anything in your replacements you will have to do a lot of routine work. Applying styles requires much more mechanical movements than in any non-WISIWIG system. In other words it's easy to start, but hard to work with. Besides, it's rather unstable. For me, a single crash with losing of one paragraph of text is enough reason not to touch this junk anymore. But most of all, there are big doubts about the "liquidity" of the **Word** files. It is claimed that you can read **Word** files with other tools like AbiWord, but in practice it's very questionable it will correctly execute all your macros, styles, and scripts. Without scripts **Microsoft Word** is nothing but just a program to write business letters. How are you going to automatically highlight the syntax of your programs, for example?

- **TEX**. That is cool, but it's an apparent overkill for me. As it was said, the full package takes about 250 megabytes. You can easily produce perfect **PDF** documents, but there's a problem with HTML. Yes, there are some tools like **LATEX2HTML** but they produce only very primitive **HTML**, and it's not possible to configure the output markup elements. Some things are possible to do with **CSS**, but not all of them. For example, if you want your **HTML** pages to be justified to a certain width, you will have to enclose every paragraph into a table. **CSS** is not that standardized to rely on, because many advanced things work differently in different browsers or don't work at all. So, the most reliable way to produce high quality **HTML** documents is to insert rather complex **HTML** tags directly into the text. Thus, another mainspring of creating **AGDoc** was to have a formatter with absolutely customizable **HTML** output. Besides, a **TEX** source (if it's good enough) looks like abracadabra in an unknown programming language. When you need to to create a good looking table, you will eventually sink in **TEX** commands. Besides, there are no ready to use tools to highlight code examples, or they require many hours of studying and tuning. However, I like the idea of syntax of **TEX** and **AGDoc** uses similar syntax, for example, `\b{Bold Text}` is displayed as **Bold Text**. I respect **TEX**, and, in fact, it's the standard to write technical articles. **AGDoc** has at least a potential possibility to produce output in **TEX** format. Well, you will need it if you want to have nice looking **PDF** along with **HTML**.

- **DocBook**. That is cool too, but it's **XML** and there's nothing to add about it. Well, I will try to add something, though :). **XML** is good as a standard for data exchange between different computer applications (with incredible overhead, though), but it has absolutely inappropriate syntax for manual editing by a human. In fact, it's as bad as naked **HTML**, despite of all possible automated editing tools. The syntax **does matter!** In **DocBook** you physically enclose sections into chapters, subsections into sections, and so on. But look into any paper book — they all have linear structure! You just read the text and do not care much about its structure, in the meaning of chapter/section/subsection…etc. Similarly, a writer wants just to **mark** a certain line as a starting point of a new subsection, but never incapsulate it into the enclosing section. In **DocBook** you have extremely long elements, that are very hard for manual handling. In general, it's a bad idea to enforce using solid elements longer than a couple of screen pages. However, despite of all disadvantages of **XML**, it might be the only appropriate solution when strict structurization is a necessity, for example, when there's a whole department working on documentation. **AGDoc** can produce **DocBook XML** too, because it's all configurable. In fact,

when debugging the **HTML** config file for this site, I achieved producing well formed **XHTML**, that can be easily read by any **XML** parser. **AGDoc** can perfectly structurize the sources and produce well formed **DocBook** format, with nested chapters, sections, subsections, etc. The task of automatic syntax highlighting can be probably solved, but it definitely requires some efforts.

- **Wiki**. It's almost what is needed because of its simple, but fair enough syntax. But not quite. First of all, I couldn't find any appropriate tools to work offline. Basically, they all provide some web-service that should be integrated on the server side. What if you just need to produce **HTML** and to upload it to your site? What if you need to have some other output format? So that, the second disadvantage is the philosophy of **Wiki** — it's hardcoded and oriented for dummies (including me, of course). And the third thing is the idea of using sequences of symbols of punctuation only can be turned into absurd. For mainstream texts that's good enough, but as soon as you need some custom formatting, like code coloring, you will have no appropriate solution, not to mention that sometimes you will want to decorate paragraphs like notes, tips, warnings, and so on. Using only empirical rules is very restrictive. Metaphorically, one can say that **Wiki** is a reciprocal of **TEX**. While **TEX** source consists of a lot of commands, sometimes looking like program code, **Wiki** source contains a lot of abracadabra that is not readable either. From this point of view **AGDoc** is a compromise between those two poles. It allows you to use **Wiki**-like elements in your text, but doesn't restrict your freedom. You can define your own formatting rules and use them in the **Wiki**-form as well as in the **TEX**-form.

- **Doxygen**. Great and deservedly popular tool. Except that it's used mostly for reverse engineering of the sources. Not many people actually put **Doxygen**-style comments into their source code. Neither do I. Why? Just because it clutters code a lot. Code must be code, comments must be comments. It means that with **Doxygen** you most likely will write one line of comments, one line of code, one line of comments, one line of code… Eventually it will look like reading two books at the same time, one line from the first, then one line from the second, and so on. When you write a C++ class with detailed description before each method, it looks even worse because you just sink in the comments (remember, you will also need to read "naked" code, without **Doxygen**). The definition of the class must be compact and observable, with comments and detailed description provided separately. Of course, you can fully describe the class right before it, but you will have to use formatting elements almost as in pure **HTML**. That breaks the whole idea. Besides, I don't think it's good to use extensive markup directly in the source files. The sources must be sources, not docs. **Doxygen** can be used to create web-sites too, but the output **HTML** is also hardcoded and cannot be configured. Another aspect is psychology. When using **Doxygen** you just describe all the classes and functions calming yourself that it's good enough. But it's not! When I see a **Doxygen** generated documentation I have no idea about the threads to pull at the beginning. In other words, it's a great navigating tool when you are familiar with ideas of the library, but not fair enough to give you a general picture. It disintegrates your sources into small details, but the common view is lost.

So that, **AGDoc** is mostly an excuse of my laziness, that however, helps me to write on-line and hard-copy documentation. I admit, I probably don't know enough about all the mentioned above documenting tools. The main rationale of **AGDoc** is that the time spent for the design and programming **AGDoc** is comparable with studying, but **AGDoc**, unlike any other tool, is always under my full control. I can develop and improve it as the case may require.

Another aspect of compatibility is file input/output and working with directories. Here **AGDoc** supports **POSIX** and **Win32**. But again, **AGDoc** doesn't need much. It uses `opendir()`, `readdir()`, `closedir()`, `mkdir()`, and `stat()` from **POSIX** and the respective `_findfirst()`, `_findnext()`, `_findclose()`, `_mkdir()`, and `_stat()` from **Win32**. So that, it's perfectly compatible with **Win32** and any **Unix** that supports very basic **POSIX**. Conditional compilation is the simplest possible one and looks like the following.

```
#if defined(_WIN32)
   . . .
   . . .   // Assume Win32
   . . .
#else
   . . .
   . . .   // Assume POSIX
   . . .
#endif
```

Another compatibility issue is that **AGDoc** does not understand drive letters and backslashes (\) in the paths. That is, all paths must be of Unix-style. **Microsoft** Windows perfectly understands Unix-style slashes in paths, so, there is no reason to support both, especially when the backslash is a key symbol in the **AGDoc** files.

to be continued...

I apologize for not providing more information, but it's in my TO DO list. For now you can try to play with the following example. First, there is an example page and its source. If it's interesting enough do the following.

1. Download:
   Windows-style Archive (http://www.antigrain.com/agdoc/agdoc.zip)
   or
   Unix-style Archive (http://www.antigrain.com/agdoc/agdoc.tar.gz)

2. Unpack and compile it. There are no compilation environments provided, but it's easy. In Windows Visual C++, just create a new console application, add all **.cpp** files, and build them. Or use the command-line compiler:
   `cl -GX *.cpp`
   On Linux or Unix with GNU C++ use
   `g++ -o agdoc -O3 *.cpp`
   … and so on.

3. When it's built, run:
   `agdoc test_project/doc_source/test.agproj`
   There new directory `test_project/output` should be created with `test.agdoc.html` and other necessary files.

4. Open `test_project/output/test.agdoc.html` and compare it with its source, `test_project/doc_source/test.agdoc`

5. Open `test_project/formatting_example.agtempl` and try to review it. I must say, I'm not an experienced **HTML** and **CSS** professional. This file is just an example that shows you the general idea of **AGDoc**. Instead of HTML tags you can put anything else, including **TEX** elements, **DocBook** elements, and so on.

6. If you like the idea and if it worth more detailed documenting, please tell me.

to be continued…