Lua2D$_{O}$X$_{lua}$

v0.1

4th July 2012

This is a hack to enable `doxygen` to document lua.
It uses the well-known `doxygen` filter mechanism to fool `doxygen` into reading foreign languages.
If this works for you then enjoy.

Contents

1 Introduction

Documenting sourcecode is a hassle. There are two basic methods of providing documented sourcecode: "literate programming" and "autodoc". Without these tools a human has to keep the documentation and code aligned, which rarely happens.

1.1 Literate Programming

If one is sufficiently organised, and it fits in with your existing work practices, then there is a lot going for literate programming. Here you write a combined document that contains both code and documentation fragments. When you want code or documentation you run it through a program which assembles the appropriate files.

This is theoretically the neatest way of making code. The best known example of literate programming is the source of TEX.

1.2 Autodoc

In the real world we get source from many sources, usually where literate programming wasn't used or isn't available.

One solution is to use an automatic documentation tool: feed it sourcecode and it will extract what documentation it can.

If you feed it raw source then all it can do is tell you where it found features such as function and class declarations.

If your source has been suitably prepared, then more detailed documentation can be produced. This preparation takes the form of "magic comments". These are source comments that contain "magic" features that let the autodoc tool know to make use of them.

1.2.1 Doxygen

`Doxygen` is one of the many autodoc tools use to document C-like languages. It is widely used and supported in the linux world and has MacOS and Windows ports.

```
83 void helloWorldOnce(){
84   printf("hello world\n");
85   }
86 /*!
87   \brief writes hello world *n
88
89   This is a demo of how to loop round
90   and print something
91
92   \param N number of times round
93 */
94 void helloWorldN(int N){
95   for (int i=0;i<N;i++)
96     printf("hello world %d\n",i);
97   }
```

void **helloWorldN**(int N)
*writes hello world *n*
This is a demo of how to loop round and print something
Parameters:
N number of times round
Definition at line **94** of file **hello.cpp**.

void **helloWorldOnce**()
Definition at line **83** of file **hello.cpp**.

1.2.2 Lua2D$_{O}$X$_{lua}$

In the case of `doxygen`, which is intended for C-like languages, it expects '`///`' or '`//!`' for C++ comments and '`/**...*/`'/'`/*!...*/`' for C-style comments. For Lua2D$_{O}$X$_{lua}$ we recycle this convention and use '`--!`' and '`--[[!...]]`'.

The internal format of the magic comments is identical to `doxygen`. This is not suprising in that `lua2dox_filter` is a `doxygen` filter. What it does is to read lua code and output something not entirely unlike C++.

This lets us make Doxygen output some documentation to let us develop this code.

```
83 function helloWorldOnce()
84   io.write("hello world\n")
85 end
86 --[[!
87   \brief writes hello world *n
88
89   This is a demo of how to loop round
90   and print something
91
92   \param N number of times round
93   ]]
94 function helloWorldN(N)
95   for i=0,N do
96     io.write("hello world" .. i .. "\n")
97   end
98 end
```

function **helloWorldN**(N)
*writes hello world *n*
This is a demo of how to loop round and print something
Parameters:
N number of times round
Definition at line **94** of file **hello.cpp**.

function **helloWorldOnce**()
Definition at line **83** of file **hello.cpp**.

1.3 Origins

Lua2D$_{O}$X$_{lua}$ inspired by the functionality of `lua2dox`[1]. Found on CPAN when looking for something else; kinda handy.

Improved from `lua2dox` to make the `doxygen` output more friendly. Runs faster in lua than in Perl.

Because this Perl based system is called "`lua2dox`", I have decided to add "`_lua`" to the name to keep the two separate, so it is called "`lua2dox_lua`".

[1] `http://search.cpan.org/~alec/Doxygen-Lua-0.02/lib/Doxygen/Lua.pm`

2 Installation

2.1 System requirements

Should run on any reasonably modern computer (PC or MacOS X).

Requires a reasonably recent: `doxygen` and `luatex` or `lua`.

2.1.1 Luatex/lua

Whilst `luatex` isn't strictly required to make it run; TEX users are more likely to have a recent version (via `texlive`) of `texlua` rather than relying on your, possibly aging, distro's lua.

It should work with either `texlua` or `lua`. The calling bash script determines which one is available (`texlua` then `lua`) and selects that one.

2.1.2 `doxygen`

Install this using whatever mechanism you usually use.

It is also helpful if you know how to use `doxygen`.

2.2 Linux/other Unices

Only tested for linux, but should work for any Unix-like OS.

For other OS you need to figure it out yourself. If you do, then please email me with the code/suggestions.

1. manually check if lua2dox.lua runs on your system.

   From the commandline run "`./lua2dox_lua --help`". If it runs and produces a sensible result then it probably works.

2. Manually pick an appropriate directory for this system.

   Make this new directory and copy/mv all these files into there. If you are installing it as a personal app then put it somewhere in your userspace. For system-wide: ask sysadmin.

3. Run "`install.sh`". This makes the links to the active code.

   Default is to put in '`~/bin`". For system-wide: ask sysadmin.

2.3 Windows

Not yet

2.4 Mac OSX

Not yet

3 Useage

Because this is a `doxygen` filter then it is assumed that the endusers are familiar with the use of this app. When familiar then one can move on to documenting lua code.

3.1 Learn to use `Doxygen`

Ensure `doxygen` is installed on your system and that you are familiar with its use. Best is to try to make and document some simple C/C++/PHP to see what it produces.

Included with this distribution are some examples of php and lua code. They are two simple programs that create some classes and then show their use. The documentation should be nearly the same as each other.

3.2 running Lua2D$_{\mathrm{O}}$X$_{lua}$

3.2.1 make Doxyfile

Run "`lua2dox_lua -g`" to create a default `Doxyfile`. This is the configuration file that `doxygen` needs for making the documentation.

Then alter it to let it recognise lua. Add the two following lines:

```
1 PROJECT_NAME      = <name of project>
2 PROJECT_NUMBER    = <version number>
3 OUTPUT_DIRECTORY  = docs
4 FILE_PATTERNS     = *.lua
5 FILTER_PATTERNS   = *.lua=lua2dox_filter
6 SOURCE_BROWSER    = yes
7 GENERATE_LATEX    = no
```

Either add them to the end or find the appropriate entry in `Doxyfile`.

3.2.2 run Lua2D$_{\mathrm{O}}$X$_{lua}$

Once `Doxyfile` has been edited run as "`lua2dox_lua`".

When reading source with classes multiple passes are needed. Each pass generates a list of member functions (as a file) that were found on this pass. This list is read in on the next pass. If the class+methods haven't changed this time then you only need to run it once, else run twice, much like running LaTeX `\tableofcontents`.

3.3 In use

As with `doxygen` the HTML output is the most useful for development. When the project is finished then PDF output (via LaTeX) may be useful.

Typically you will have some source files open in your editor/IDE. Also keep a web-browser window open pointing at the documentation.

After a bit of editing the documentation will become stale. At this point run Lua2D$_{\mathrm{O}}$X$_{lua}$ once/twice and refresh the HTML pages. Then your documentation will be updated.

4 Design

Lua2D$_{\mathrm{O}}$X$_{lua}$ is a `doxygen` filter. It has a basic parser that will read each line of lua and output an equivalent in pseudo-C++. It loops round until it has found the end of the file. It only has to be good enough for `doxygen` to see it as legal. Therefore our lua interpreter is fairly limited, but "good enough".

The parser is relatively crude and assumes that the programmer writes lua in a fairly clean way. Some bits of lua will confuse it.

In order for the source browser to work in the `doxygen` output it must preserve the number of lines. The output of each line is either written as-is or a converted line. When `doxygen` re-reads the source to make the source-HTML/LaTeX/... it uses the line number to set the position.

It only supports comments that preceded the function/class declaration. Any in the middle will act as cruft in the resulting documentation. This will be slightly out of place but at least should refer to somewhere near to where it was intended. In particular "`@todo`". This will appear in the ToDo list, but not associated with the right function.

4.1 Classes

Class declarations are assumed to be using a "well-known" userdefined function "`class()`". It processes

"`AA = class(A)`" → "`class AA: public A{};`".

However it will probably have some member functions. These get converted from

"`A.foo(a,b)`" → "`A::foo(a,b)`". This is stored in a temporary file. When Lua2D$_{\mathrm{O}}$X$_{lua}$ is run a subsequent time this is used to generate a list of methods. So

"`AA = class(A)`" → "`class AA: public A{foo(a,b);bar(x);};`"

4.2 Multiline function declarations

Because Lua2D$_{\mathrm{O}}$X$_{lua}$ can only process one line at a time it cannot correctly process function declarations with multiline parameter list. This is because the parser isn't sophisticated enough to guarantee finding the close paren on some random later line.

So I have put in a hack that will insert the "missing" close paren. It looks at the function declaration that you have got and checks for a close paren. It if is not there then it will add a spoof parameter and the close.

The effect is that you will get the function documented, but not with the parameter list you might expect.

```
94 function foo(A
95    ,B
96    ,C)
```

Becomes

```
94 function foo(A,___MissingCloseParenHere___)
```

4.3 Multiple magic comments

The hacking of lua to C++ by Lua2D$_{\mathrm{O}}$X$_{lua}$ has the limitation that some magic comments will not get associated with the correct function/class. This is a "feature" that might get correct in a later version.