LUAMETAFUN

the new interfaces

context **2021** meeting

Three subsystems

- The core of the engine in still TEX. It all starts there.
- The TEX internals are opened up via Lua. We can call out to Lua from TEX and to some extend from Lua to TEX. Quite often we push back something via the input channels.
- The MetaPost library is accessed from Lua. So from within TEX there is always Lua in between. Results go back via Lua. The library can also call out to Lua and from there to TEX.
- This means that all three major components of LuaMetaTEX can talk to each other and use each others capabilities.
- With Lua in the center, we also have access to other mechanism, for instance fonts, graphics and libraries.
- In ConTEXt the Lua language also permits using xml, json, csv, sql databases and other input that can be dealt with programmatically.
- All has been reasonably optimized for efficiency and performance.

The (LuaTEX) library

- Turning MetaPost into a library has been a subproject of the LuaTEX project. The semi-official team (Taco, Jacko, Hans, Luigi) got John Hobbies blessing.
- This was a rather massive (and impressive) operation by Taco because multiple number models were to be supported and the internals had to be made such that different backends were possible. All with remaining perfect (DEK) compatibility.
- The MetaPost library serves both the stand alone program and LuaTEX.
- That means the PostScript backend is built in plus some basic (Type1) font handling. We support pdf output via the MetaPost Lua backend (in MkII that is done by parsing the PostScript and specials).
- In addition there is png and svg output. It helps that MetaPost output is rather simple.
- The LuaTEX engine uses the Lua backend which represents the result in Lua tables resembling the MetaPost internal representation.
- The library supports scaled and double (internal) but also binary and decimal number models that use (linked in) libraries.

The (LuaMetaTEX) library

- We don't need the PostScript backend (which only does Type1 anyway).
- We also have no use for svg and png output.
- The binary number model has no advantages over the decimal one but brings quite some dependency with it (library code).
- The Type1 font support is not used in ConTEXt because we handle text differently.
- All this means that we can do with a smaller (simplified) MetaPost library.
- The codebase has been overhauled. We still have `.w` files (cweb) but use a Lua script to convert that to C which means that we have better control over how it comes out.
- As with LuaTEX the file io, message handling etc. now largely goes via Lua; it is more integrated.
- The same is true for scanning interfaces and return values (injectors). That also made for more symbolic coding.
- Memory management (allocation) is under engine control (as with TEX and Lua); we use a common high performance allocator library.

The (LuaMetaTEX) library

- Some already present mechanism have been extended, for instance clips have pre- and post-scripts.
- A grouping wrapper has been added (handy for some graphic trickery supported in the back-end.)
- The `runscript` primitive supports symbolic references to functions (of course to be provided at the Lua end).
- The `runscript` return values can be more native, in addition to the already present (default) `scantokens` support.
- Internals are extended with booleans and strings.
- Output (paths, clips etc) can be stacked in a different order.
- There are additional statistics available.
- In some places performance could be improved.
- In the meantime it can be considered a major upgrade and (for various reasons) backporting to LuaTEX makes no sense. And yes, all errors are mine.

The Luafication

See Taco's presentation where he gives some examples.

Callbacks

We need to hook in some functions:

	`find_file`	`(name,mode,kind)`	locate a file (usually within the tds setup)
`f`	`open_file`	`(name,mode,kind)`	open given file
	`close_file`	`(handle)`	close opened file
`s`	`read_file`	`(handle,size)`	read from file
	`write_file`	`(handle,str)`	write to file
`s`	`run_script`	`(code,size,index)`	run the given string as Lua script
`s`	`make_text`	`(str,size,mode)`	process btex/etex
	`run_internal`	`(action,index,kind,name)`	act on internal definition
`n`	`run_overload`	`(property,name,mode)`	process overload check
	`run_logger`	`(target,str,size)`	process log message
	`run_error`	`(message,help,interaction)`	handle error (message)
	`run_warning`	`(message)`	handle warning

Two calling methods

The runner can be called as:

```
runscript("mp.MyFunction()")
```

which implies at the Lua end:

```
function mp.MyFunction()
    ...
end
```

Here the callback function is responsible for loading the string and executing it.

Alternatively one can say:

```
runscript <number>
```

The number can be intercepted at the Lua end to do some associated action.

Variables

We can do:

```
lua.mp.MyFunction("foo",123,true)
```

which in the end is equivalent to:

```
runscript("mp.MyFunction('foo',123,true)")
```

Alternatively one can pick up values by scanning: like `scannext`, `scanexpression`, `scantoken`, `scansymbol`, `scannumeric`, `scaninteger`, `scanboolean`, `scanstring`, `scanpair`, `scancolor`, `scancmykcolor`, `scantransform`, `scanpath`, `scanpen`, etc.

Return values

The runner can return:

- a string that gets fed into the `scantokens` primitive
- a numeric or boolean that gets injected as native MetaPost object
- a table that gets concatenated and fed into the `scantokens` primitive
- `true` and a second argument that gets converted into a native MetaPost object
- in the last case the number of table elements determines the object

Instead of returning a value one can inject: `injectnumeric`, `injectinteger`, `injectboolean`, `injectstring`, `injectpair`, `injectcolor`, `injectcmykcolor`, `injecttransform`, `injectpath`, `injectwhatever`, etc. and these accept one or more values and/or tables.

These mechanisms might evolve a bit over time. Lots of examples can be found in the `mlib-*.lmt` files.

Parameters

- The new interfaces permit us to program quite robust parameter driven interfaces that (sort of) match the way we do things at the TEX end.
- The distribution has several examples of usage and more will be added.
- Macros that use the new mechanisms can be recognized by the `lmt_` prefix.

```
lmt_mytrick [
    somestring  = "test",
    somenumeric = 123,
    someboolean = true,
    somecolor   = (1, 0, 1),
    somepath    = fullsquare scaled 10cm,
    somelist    = { (0, 0), (1, 3), (8, 9) },
    sometable   = [
        somenumeric = 321,
   ],
] ;
```

Show the pattern of defining these at the Lua end and in MetaPost files.