FOSDEM'26

Université Libre Bruxelles / 1 February

Building a minimal cross-platform terminal UI library

Thijs Schreijer

Source code:

- https://github.com/lunarmodules/luasystem
- https://github.com/lunarmodules/terminal.lua

GSoC 2025

Google Summer of Code

Why another terminal UI library?

- implementing a UI is hard;
  - massive APIs,
  - multi-threading,
  - Etc.
- a simple way to build a useful app, to get started: cross-platform

- as Copas maintainer; non-blocking coroutine behaviour

Demo

- Luarocks
- Luarox
- Luarocket

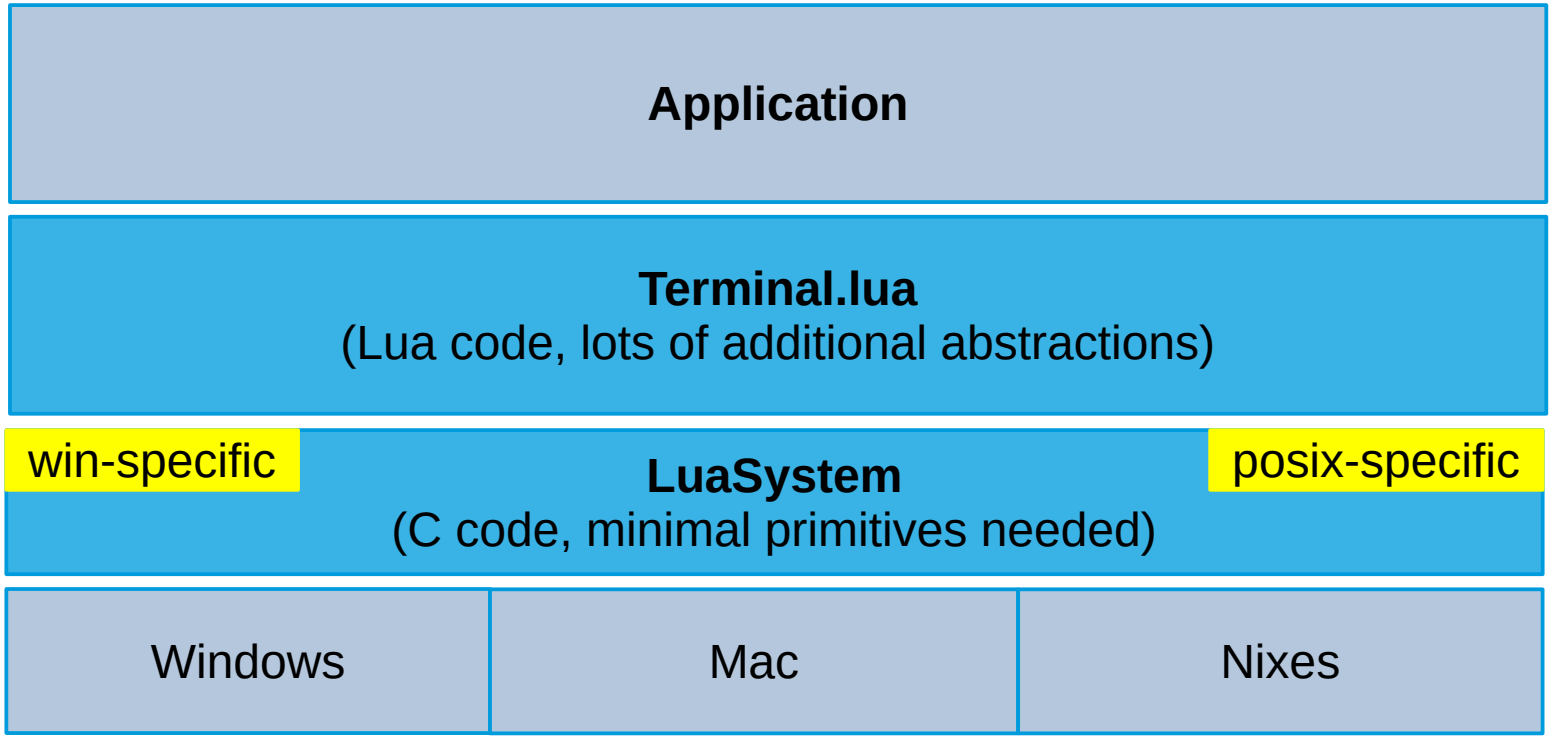
Goals

- Cross-platform
  - Posix / Mac / Windows
- non-blocking input
- mechanisms over policies
  - Don’t force event mechanisms and the like
- no external dependencies
  - Too hard for Windows users
- minimize platform specifics

non-goals

- mouse support
- overlapping windows (screen buffers etc)

Stack


Application
Terminal.lua
(Lua code, lots of additional abstractions)
win-specific
LuaSystem
(C code, minimal primitives needed)
posix-specific
Windows
Mac
Nixes

The easy/common part

- ANSI sequences for modern terminals
  - available in Windows 10 since 2019 (Windows Terminal app)
  - no support for `cmd.exe`!
- Getting terminal size (all OS'es have API calls)

Challenges: UTF-8/Unicode

- Non-native for Windows
  - Output is easy; ANSI sequences to `stdout`
  - Input not
- Multibyte encoding (Lua strings are byte-arrays)
- **Single/double/ambiguous display-width**
  - Display-width API is unavailable on Windows
  - Used an opensource implementation by Markus Kuhn; `wcwidth`
  - Ambiguous-width still problematic

Challenges: UTF-8/Unicode

- Implemented width checking:
  - `utf8cwidth(char)`
  - `utf8swidth(string)`
- Implemented `` `string.sub` `` equivalents:
  - `utf8sub(str, i, j)`
  - `utf8sub_col(str, i, j[, no_pad=false])`

Challenges: UTF-8/Unicode

- EditLine object:
  - Holds a string
  - Manipulate using a cursor
  - Tracks position and sizes both in characters and in display columns

API examples:

```
editline:delete_word ([n=1])   Delete until the end of the current word.
editline:format ([opts])       Format the contents for display, (word)wrap.
editline:goto_home ()          Moves the cursor to the start of the string.
editline:insert (s)            Inserts a string at the current cursor position.
editline:left ([n=1])          Moves the cursor to the left.
editline:left_word ([n=1])     Moves the cursor to the start of the current word.
editline:pos_char ()           Returns the current cursor position (chars).
editline:pos_col ()            Returns the current cursor position (columns).
editline:sub_char ([i=1[, j=-1]])  Returns a new Editline object being a substring.
```

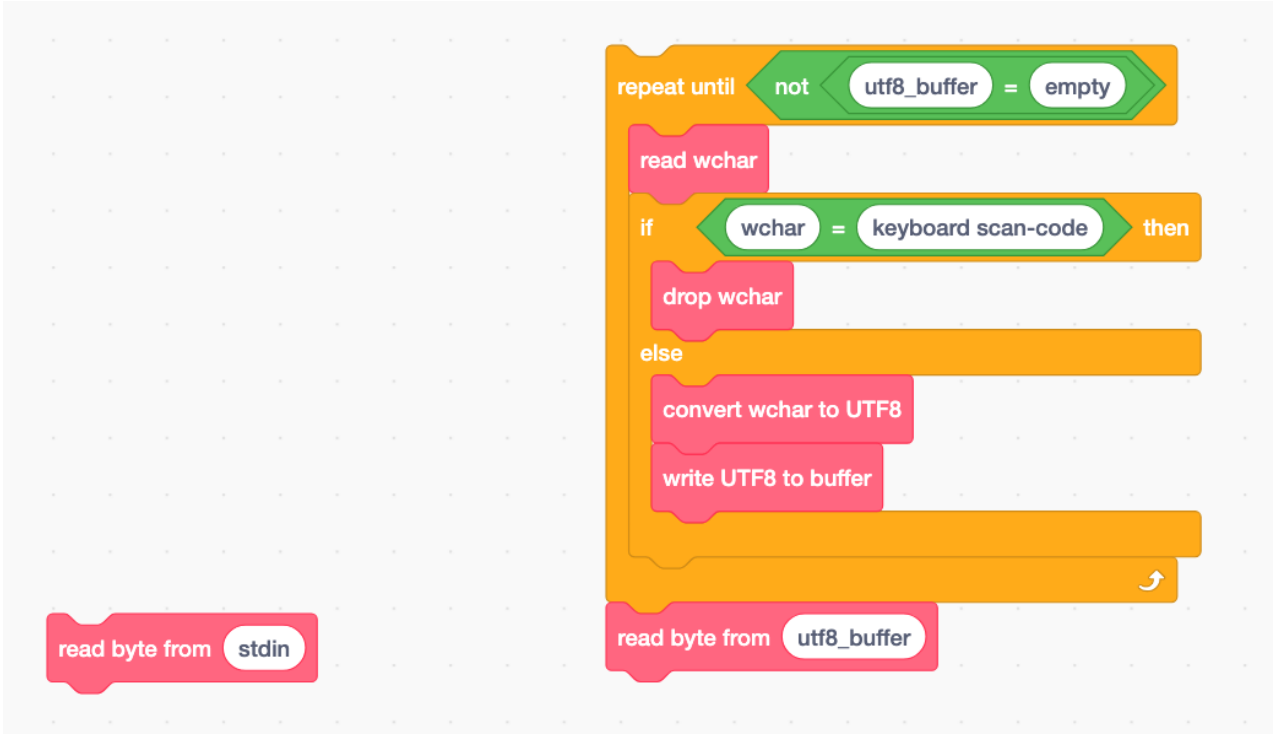
Challenges: keyboard input

- There is no non-blocking `stdin` reading on Windows
- Fallback to `conio.h` (reading directly from keyboard buffer)

Challenges: keyboard input

Posix

Windows



C code in LuaSystem

Initialization

- Posix
  - Disable canonical mode
  - Disable echo
  - Detach FDs
  - Set `stdin` to non-blocking
- Windows
  - Set console output codepage to 65001 (UTF8)
  - Enable virtual terminal processing (on in- and output)

Challenges: non-blocking

3 functions based on reading individual bytes

```
key, err = _readkey()                          -- internal C method, read single byte

key, err = readkey(timeout, [sleep_func])      -- read single byte, with timeout

seq, type = readansi(timeout, [sleep_func]) -- read utf8-char/ANSI sequence
    -- seq = sequence read (string)
    -- type = any of: "char", "ctrl", "ansi" (string)
```

By changing the default `sleep` method any keyboard-code becomes non-blocking

Querying is slow

- Terminal takes time to respond to a query
- Sleeping to wait takes 2 to 15 ms typically, occasionally 100ms
- Track state instead
- Implemented declarative stacks to track:
  - Cursor shape
  - Cursor visibility
  - Scroll-region
  - Text attributes (color, brightness, underline, reverse)

Stack example

```
t.text.stack.push{          -- Push new attributes
  fg = "red",
  brightness = "bright",
}

t.output.print("Hello bright red World!")

t.text.stack.pop()          -- Restore text attributes
```

API usage

POSIX API Calls

- `isatty()` - Check if file descriptor is a TTY
- `tcgetattr()` - Get terminal attributes
- `tcsetattr()` - Set terminal attributes
- `fcntl()` - File control operations (F_GETFL, F_SETFL for O_NONBLOCK)
- `ioctl()` - I/O control (TIOCGWINSZ for terminal size)

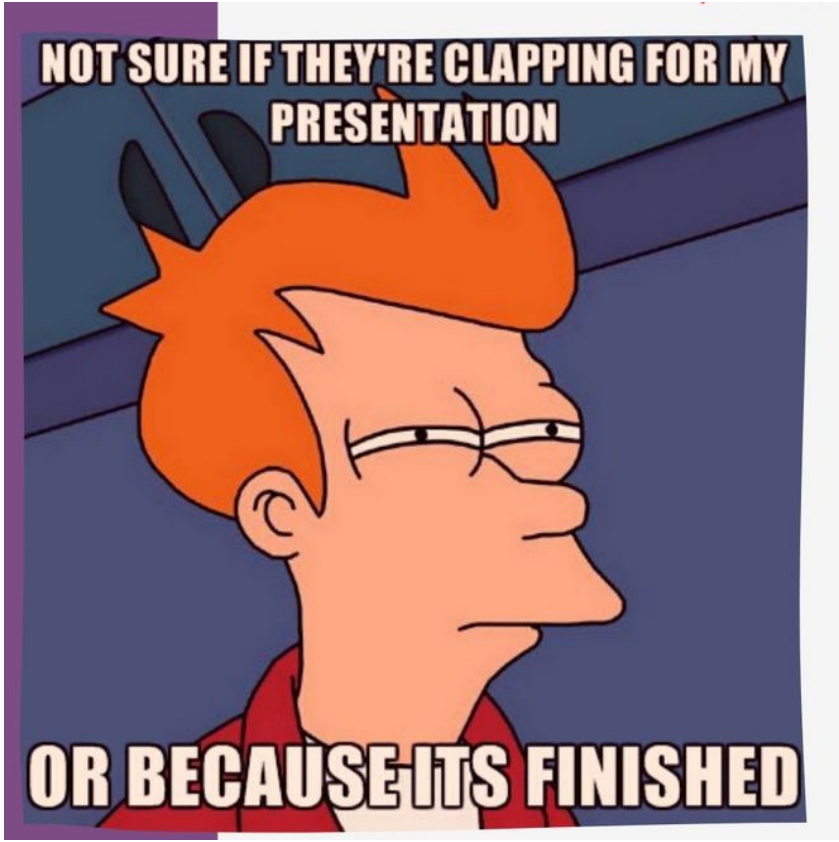
Windows API Calls

- `isatty()` - Check if file descriptor is a TTY
- `Get/SetConsoleMode()` - Get/set console mode flags
- `GetConsoleScreenBufferInfo()` - Get terminal/console dimensions
- `Get/SetConsoleCP()` - Get/set console input code page
- `Get/SetConsoleOutputCP()` - Get/set console output code page
- `_kbhit()` - Check if a key was pressed (non-blocking)
- `_getwch()` - Read a wide character from input

Concluding

- Result quite powerful with minimalistic primitives, no dependencies
- Uniform keyboard reading
  - Ancient `conio.h` …
  - Non-blocking
- Display width is painful
  - `wcwidth` implementation
  - Need string manipulation based on display colums
  - EditLine object for higher-level manipulation and formatting
- Stacks to track terminal state (querying is slow)

Questions?


NOT SURE IF THEY'RE CLAPPING FOR MY PRESENTATION
OR BECAUSE ITS FINISHED