

Hacking the Planet (with Notcurses)

A Guide to TUIs and Character Graphics

Nick Black, Consulting Scientist
nickblack@linux.com

March 24, 2020



Update, 2021-06-29: This edition was completed in March 2020. Since then, great swaths of the Notcourses API have changed. Many if not most of the examples and references in this text are no longer completely accurate. Read with caution! I'll update it when I get the chance, ideally by the end of 2021.

nick black aka dank

For T. S. Eliot, il miglior fabbro.

For Jeanette Martin, for exhortations to go H.A.M.

For Jim Greenlee, for speaking rigor to my programming.

For Prof. Hyesoon Kim, for introducing me to the glorious world inside the die.

For Prof. Richard Vuduc, for demonstrating serenity in brilliance, and kindness in dominance.

...but mostly for Emily.

Contents

List of Figures	vi
List of Listings	vii
List of Tables	ix
Foreword	xi
¡Peligro!	xi
Errata	xi
1 Introduction	1
2 Right, what’s all this, then?	3
3 Using direct mode with standard IO	6
3.1 Example: presenting <i>House of Leaves</i>	7
3.2 Example: colorizing a dumb game	8
3.3 Advanced coëxistence with stdio	9
3.4 Use in multithreaded environments	10
4 Using fullscreen mode	12
4.1 The notcurses_options structure	13
4.2 Functions on notcurses objects	15
4.3 Reading, rendering, rasterizing, and writing	15
4.4 Capabilities	19
4.5 Statistics	19
4.6 Use in multithreaded environments	21
5 A simple notcurses render/event loop	22
5.1 Example: moving tetriminos with a keyboard	22
6 Terminal mechanics	34
6.1 Terminals and the UNIX process model	36
6.2 Control sequences ANSI and otherwise	38
6.3 Terminfo	39
7 Character encodings and glyphs	41
7.1 Everyone loves ASCII	42
7.2 Octa- and hexabit character sets messily diverge	43
7.3 Consilience: the Universal Character Set	45
7.4 Fixed-width fonts ain’t so fixed	46
7.5 Emoji	47
7.6 Stupid Unicode tricks	49
7.7 UTF-8	52
8 Using ncplanes	55
8.1 Moving and resizing planes	56
8.2 Cells	58
8.3 egcpools	60
8.4 Alpha blending and plane transparency	62
8.5 Manual palette-indexed color	63
8.6 Fading and pulsing planes	63
9 Writing and styling text	69

9.1	Writing text to planes	69
9.2	The 32-bit attribute value	72
9.3	The 64-bit channels value	74
10	Lines, boxes, and fills	81
10.1	Linear interpolation (“lerping”) and lines	81
10.2	Boxes	82
10.3	Gradients and polyfills	83
10.4	Blitting	83
10.5	Staining	84
11	Multimedia (images and videos)	88
11.1	Streaming video/animated GIFs.	89
11.2	Scaling images and video	89
11.3	Sprites	90
12	Collecting and dispatching input	92
13	UI widgets	94
13.1	Selectors and multiselectors	94
13.2	Menus	94
13.3	Reels	95
13.4	Example: let’s rip off whiptail	101
14	Complex examples	105
14.1	Example: walking through notcurses-demo	105
14.2	Example: let’s rip off tetris	114
Appendix A	A brief history of character graphics	125
A.1	The DEC VTxxx terminals and ANSI X3.64-1979	128
A.2	The Curses API	129
Appendix B	Wherein shade is thrown at terminal emulators	130
Appendix C	The Linux console	137
Appendix D	Unicode 13	138
Appendix E	Relevant Standards	139
	Glossary of terms	140
	References	146
	Acknowledgments	151
	About the author	152

List of Figures

3	NCURSES TUIs: Ncmpepp and Omphalos.	4
4	Non-NCURSES TUIs: Mapscii and Growlight.	4
5	Put not your trust in hackers making a fetish of Xerox PARC.	5
7	hol-formatter as run on OCRd input.	7
8	Colorized output from hilodirect.c.	11
9	Notcurses refusing to start due to an unsupported character encoding.	13
10	Inhibiting use of the alternate screen.	14
11	Notcurses initialization warnings.	15
12	Margins can be used around the rendering area.	16
13	Piping hot tetriminos, fresh from <i>Spiritus Mundi</i>	22
14	Font aspect ratios center around 0.5.	23
15	Unspeakably foul.	25
16	Adding a gradient.	25
17	Linear expansion.	25
18	Trigonometry!	28
19	Unicode Blocks.	28
20	Better blocks.	28
21	Adjusting for cell aspect ratio.	29
22	Undesirable plane interaction.	29
23	Resolve it with transparent planes.	29
24	Background image, greyscaled.	32
25	Opaque highlight box.	32
26	Rotation thread enabled.	32
27	A serial console, from hardware to userspace.	35
28	Three different Linux framebuffer implementations.	35
29	VESA + USB virtual console, from hardware to userspace.	37
30	Terminal-connected processes on a Linux machine.	38
31	Processes, sessions, PPIDs, and TTYs.	40
32	The legendary Code Page 00437.	44
33	Timeline of selected sets.	44
34	2020's Unicode 13.0 ships 143,859 character definitions.	45
35	Charsets served over the last decade on the Web (<i>source: W3Techs</i>).	46
36	Some very wide Unicode glyphs.	48
37	Unicode 13.0 Block Elements (<i>source: Antonsusi under CCA3.0</i>).	49
38	Unicode Braille characters.	50
39	A pangram using a variety of Unicode texts.	51
40	Some of “Eli the Bearded”'s pseudoalphabets from http://qaz.wtf/u/	51
41	Past, present, future, all are one in Yog-Sothoth.	52
42	Flow of control characters through historic standards.	54
43	Unicode box-drawing characters (<i>source: Chininazu12, public domain</i>).	81
44	Naked selector.	94
45	Selector with a long title.	94
46	Short title intersecting with header.	95
47	Selector with a long header.	96
48	Selector with a long footer and no header.	96
49	Multiselector.	97
50	Menu along the top of the standard plane.	98
51	Menu along the bottom of the standard plane.	98
52	WarMECH and a translucent menu.	99
53	growlight, a program built around reels.	99
54	Inspecting the terminfo database.	105
55	“Intro”.	106

56	“X-Ray”. Very large planes.	106
57	“Eagle”, first phase. Parallax scrolling on large image.	106
58	“Eagle”, second phase. Sprites. Zoomed image.	106
59	“Trans”, early phase.	107
60	“Trans”, middle phase.	107
61	“Trans”, late phase.	107
62	“Highcon”. High-contrast text.	107
63	“Grid”. Max RGB density.	107
64	“Box”. Lerped perimeters. Precise Unicode. Color sweeps.	108
65	“Sliders”. Partial fades. Animation. Gradients.	108
66	“Reels”. The ncreel widget.	108
67	“Whiteout”. Translucency.	108
68	“Chunli”. Sprite animation.	109
69	“Chunli”. Sprite animation.	110
70	“Uniblock”. Hangul syllables.	111
71	“Uniblock”. Emoji.	111
72	“View”. Scaling an image.	111
73	“View”. Transparent images.	111
74	“View”. Streaming video with high-contrast text.	112
75	“View”. Notice the high-contrast kicking in.	112
76	“Jungle”. Palette-indexed image.	112
77	“Fallin’”, early phase.	113
78	“Fallin’”, late phase.	113
79	“Luigi”. Multiple sprites.	113
80	“Outro”. Fades atop video.	113
81	Tetris—primed to score.	117
82	Tetris—boom!	117
83	Tetris::MoveRight() and Tetris::MoveLeft().	117
84	Tetris::RotateCcw() and Tetris::RotateCw().	118
85	Dumb terminals of the 1970s.	126
86	Digital Equipment Corporation terminals of the 1970s and 1980s.	126
87	VT220 glyph dump.	126
89	Odd performance from xterm.	131
90	Flame graph: alacrity+notcurses-demo.	132
91	Flame graph: xterm+notcurses-demo.	132
92	Intel i7-8550U benchmarks, varying widths.	133
93	NVIDIA GTX 1080 benchmarks, varying widths.	134
94	Bytes output per demo per term.	135
95	80x52 Intel i7-8550U benchmarks.	135
96	382x74 NVIDIA GTX 1080 benchmarks.	136

List of Listings

1	Initializing and stopping direct mode.	6
2	The ncdirect styling API.	6
3	Geometry discovery with ncdirect.	7
4	Cursor management with ncdirect.	7
5	hol-formatter.c, a streaming formatter.	8
6	hilostdio.c, a simple guessing game.	9
7	hilodirect.c, a colored version of the guessing game.	10
8	Initializing and stopping fullscreen mode.	12
9	Essential functions on notcurses objects.	17
10	Dealing with external events.	17
11	Rendering syncs the physical display to our visual planes.	17

12	The capabilities API.	20
13	The statistics API.	21
14	The seven canonical tetriminos (from <code>tetrimino.c</code>).	23
15	Creating a single tetrimino (from <code>tetrimino.c</code>).	24
16	Distributing the tetriminos with “flying-v” technique (from <code>tetrimino.c</code>).	25
17	A one-shot, display-only <code>main()</code> (from <code>tetrimino.c</code>).	25
18	Switching between pieces (from <code>tetrimino-input.c</code>).	26
19	Trigonometric layout: simpler, yet more accurate (from <code>tetrimino-input.c</code>).	27
20	Core input dispatch (from <code>tetrimino-input.c</code>).	28
21	Improving appearance with Unicode Block Elements (from <code>tetrimino-input.c</code>).	29
22	Throwing in a background (from <code>tetrimino-input.c</code>).	30
23	Marshaling structure for shared state (from <code>tetrimino-input.c</code>).	30
24	Spin them doggies (from <code>tetrimino-input.c</code>).	31
25	Set the selection off with a coaster (from <code>tetrimino-input.c</code>).	32
26	Putting it all together (from <code>tetrimino-input.c</code>).	33
27	Creating a new plane aligned relative to another.	55
28	Duplicating a plane.	56
29	Manipulating a plane’s user pointer.	56
30	Destroying planes.	56
31	Moving planes on the z axis.	57
32	Moving planes on the x and y axis.	57
33	Resizing a plane can retain any amount of the old material.	57
34	Translating coordinates between planes.	58
35	Reflecting on the plane to acquire its contents.	58
36	Accessing a plane’s raw channels and attributes.	58
37	Manipulating a plane’s active foreground channel.	59
38	Manipulating a plane’s active background channel.	60
39	Manipulating a plane’s active attributes.	60
40	Manipulating a plane’s base cell.	61
41	Aligning output within a plane.	61
42	Rotating planes.	61
43	The <code>cell</code> definition.	61
44	Modifying <code>cell</code> channels.	62
45	<code>cell</code> foreground RGBA functionality.	63
46	<code>cell</code> background RGBA functionality.	64
47	<code>cell</code> palette-indexed color functionality.	65
48	<code>cell</code> default color functionality.	66
49	The <code>palette256</code> API facilitates manual palette programming.	67
50	Palette fades.	68
51	Callback type for pulsing and fading.	68
52	Cursor management. Each plane has its own cursor.	69
53	<code>mbswidth()</code> counts columns in a multibyte string.	69
54	Output of <code>cells</code> to planes.	70
55	Direct output of single-byte UTF-8 to planes.	70
56	Direct output of a single <code>wchar_t</code>	70
57	Output of single EGCs to planes.	71
58	Output of single <code>wchar_t</code> -encoded EGCs to planes.	71
59	Output of strings to planes.	72
60	Output of wide strings to planes.	72
61	Formatted output to planes.	73
62	Bits of the <code>channels</code> type	74
63	Masks and other constants for working with channels	74
64	<code>cell</code> predicates for testing multicolumn properties.	75
65	The full channel API.	76

66	The full channels API.	80
67	Channel blending.	80
68	Functions for drawing lines.	82
69	Functions for drawing rectilinear boxes.	83
70	Helpers for rounded-corner boxes.	84
71	Helpers for doubly-thicc boxes.	85
72	Polyfills and plane erasure.	85
73	Drawing gradients.	86
74	Blitting BGRx and RGBA.	87
75	Changing attributes or channels in isolation.	87
76	Opening and destroying multimedia with ncvisual.	88
77	Decoding and rendering multimedia with ncvisual.	88
78	Acquiring subtitles.	89
79	streamcb callback type and ncvisual_simple_streamer().	89
80	Media streaming and ncvisual_simple_streamer().	90
81	Scaling media onto a new plane.	90
82	Input can be acquired in nonblocking, blocking, or timed fashion.	93
83	Mouse events must be explicitly enabled, and can be disabled.	93
84	Selector creation.	95
85	Selector control.	100
86	Menu creation.	100
87	Menu control.	101
88	Reel creation.	102
89	Tablet redraw callback function type.	103
90	Reel control.	104
91	Tetris helpers Gravity() and StainBoard().	115
92	Tetris::LineClear().	115
93	Drawing the background and the gameplay plane.	119
94	Tetris::InvalidMove().	120
95	Tetris::MoveDown().	121
96	Tetris::Ticker().	121
97	Tetris::LockPiece().	122
98	Tetris::NewPiece().	123

List of Tables

1	RS-232/EIA-232 pin mappings	34
2	Expanded contents of /proc/tty/drivers	36
3	ANSI (actually ECMA-48) escape codes	39
4	ANSI X3.4-1986 (ASCII)	42
5	Usual UNIX semantics of C0	43
6	The ten Unicode design principles.	45
7	The six named UCS planes.	46
8	Flags in Unicode.	49
9	Just a few of Unicode's many cyclic groups.	50
10	Transition matrix for alpha blending.	66
11	The NCSTYLE bits.	73
12	ctlword parameter for box-drawing.	82
13	Relevant terminfo properties.	105
14	Some historical terminals and their resolutions.	125
15	Benchmarking properties of various demos.	130
16	Terminal software used for benchmarking.	131
17	Unicode 13.0.0 Standard Annexes and Synchronized Technical Standards.	138

18 ECMA/ANSI/ISO standards referenced in this text. 139

Foreword

Hacking the Planet with Notcurses: A Guide to Character Graphics and TUIs. Copyright © 2020 Nick Black. ISBN: 9798620069491

This edition corresponds to version 1.2.4 of the Notcurses library, released 2020-03-24. Notcurses can be downloaded from <https://github.com/dankamongmen/notcurses>. This document can be downloaded from <https://nick-black.com/htp-notcurses.pdf>.

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this document except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The entirety of this work is Free Documentation, written for love and released to instruct. If you’d like to show thanks for my efforts, I encourage a donation to the [Foundation for Individual Rights in Education](https://www.thefire.org/) (<https://www.thefire.org/>). Alternatively, buy the paperback!

This work was prepared on a Debian Unstable Linux workstation and an Arch Linux laptop, using Vim, X_YLA_TE_X, and the GIMP. A FreeBSD 12 machine was emulated with QEMU.

Tetris © The Tetris Company, LLC. *Hackers (1995)* © United Artists Pictures. *House of Leaves (2000)* © Penguin Random House. “Ruins with Rain” © Mark Ferrari/Living Worlds. “Final Fantasy” © Square Enix Co Ltd. “Super Mario Bros.” © Nintendo of America. “Ninja Gaiden” © Koei Tecmo America. “Street Fighter II” and “Mega Man 2” © Capcom of America. Please don’t sue me.

¡Peligro!

The code written for this book attempts to minimize use of vertical space (fewer pages → cheaper book) without eliding error checking (or crossing into the realms of the grotesque). Error handling is a fundamental slog of C programming, one that inevitably complicates reliable applications.

These listings cannot be considered examples of good general style...but they *do* get the job done.

Three irregular idioms show up frequently:

- Use of `|=` to collect non-zero return values from each of a series of non-interdependent function calls.
- Right-hand-side conditionals fed into `|=`, e.g. `r |= (printf("dank") < 0);`.
- Extensive use of `||`’s short-circuiting property.

Errata

A list of errata for this First Edition is kept at

https://nick-black.com/dankwiki/index.php/Hacking_The_Planet!_with_Notcurses

Please contact me with corrections, either via mail at nickblack@linux.com, or via PR on

<https://github.com/dirty-south-supercomputing/technicalreports>

Our fine arts were developed, their types and uses were established, in times very different from the present, by men whose power of action upon things was insignificant in comparison with ours. But the amazing growth of our techniques, the adaptability and precision they have attained, the ideas and habits they are creating, make it a certainty that profound changes are impending in the ancient craft of the Beautiful.

Paul Valéry

1 Introduction

I implemented Notcurses in the winter of 2019 after having a few patches rejected from NCURSES. The first commit was pushed 2019-11-16. It proved to be seductive as hell, and it was only with difficulty that I tore myself away following three months of hard work. I started writing this manuscript 2020-02-12, following the 1.1.8 release. By that time, Notcurses subsumed large chunks of NCURSES, adding a great deal more. The project had three major goals:

- to provide NCURSES-like functionality with 24-bit color, safety in the presence of multithreading, and full Unicode support,
- to reduce the amount of boilerplate code necessary for the UIs of my TUI applications, including *growlight* and *omphalos*, and
- to portably facilitate the most vivid character graphics possible.

Many people asked how such a thing was useful. My usual response was that numerous devices don't present a bitmap interface, that X11 GUIs run remotely over SSH are effectively unusable beyond a local network, that plenty of machines don't have a GUI environment installed, that there are obvious applications for large outdoor displays, and that Sixel isn't well-supported across different terminal emulators. It seems impossible in an age of gigatransistor graphics cards, but the text environment still presents perceivably less latency than most GUI toolkits. That I was able to remove thousands of lines of NCURSES code from my applications was a nice side benefit.

In truth, the main reasons were that it was fun, and I wanted to see how far I could push it.

As I write this, Notcurses is present in Arch's AUR, and is awaiting promotion from the Debian Incoming queue. Written as a C core, it enjoys C++, Python, and Rust wrappers. I have submitted it as a backend to NESTopia and RetroArch, and intend to integrate it into Mesa as an OpenGL backend. So long as one can live with the limited resolution available when a screen is divided into rectangular cells, it can handle any graphics thrown at it. I hope to see it displace NCURSES as the go-to character graphics library for new applications (there is little value in porting existing applications to Notcurses, since an unchanged application wouldn't take advantage of its advanced features).

While the X/Open Curses specification is unlikely to ever go away (nor should it, as a lowest-common-denominator interface to devices Notcurses is unlikely to ever support), I believe Notcurses to present a superior interface and implementation for modern TUI applications.

The console ain't dead! Hack on, hax0rs.

—February–March 2020, Atlanta

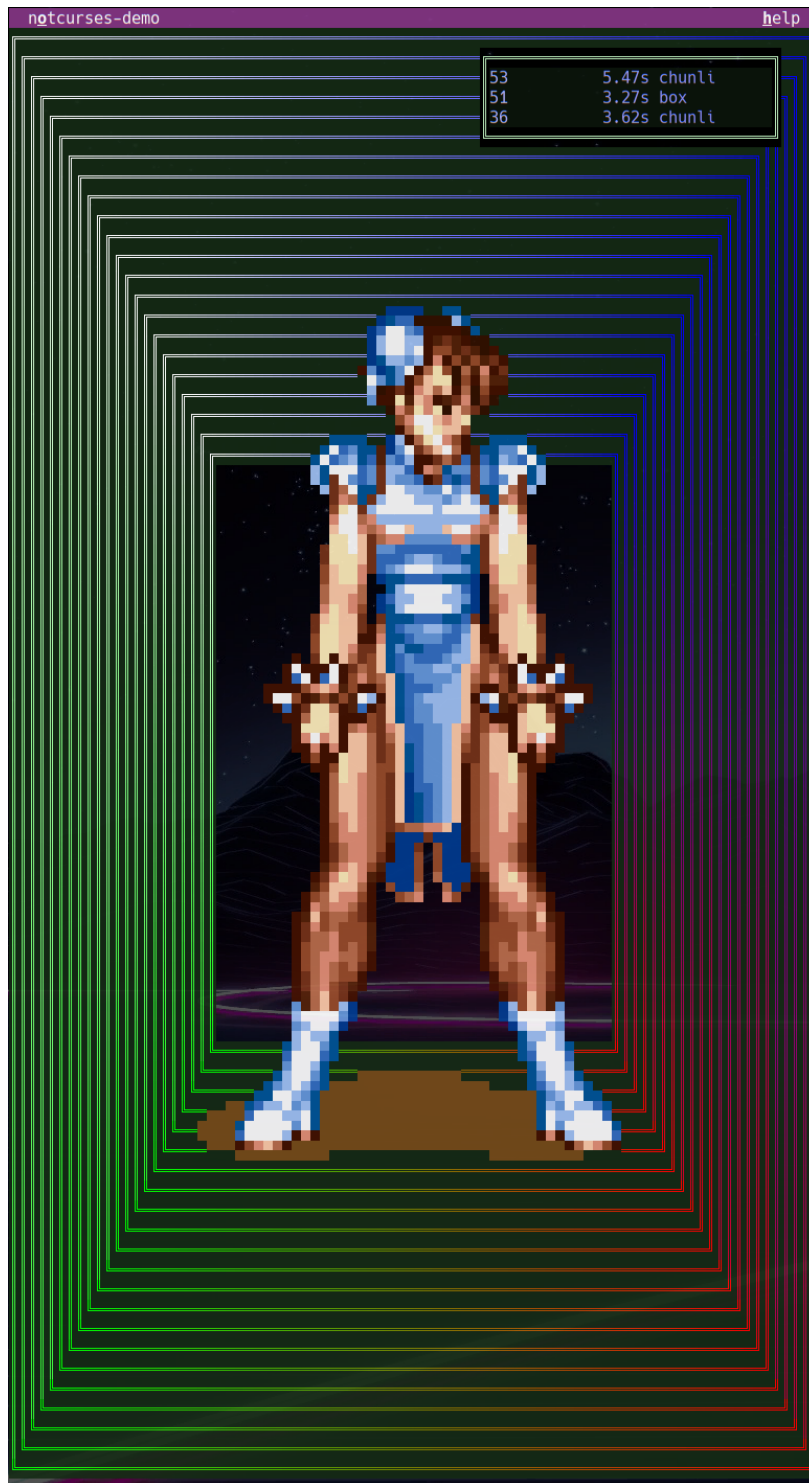


FIGURE 2: A rendered scene from the “chunli” demo (see Chapter 14.1), using some of the advanced capabilities of Notcurses. The Chun-Li sprite has been loaded from a transparent PNG (Chapter 11) atop boxes (Chapter 10.2) drawn using Unicode (Chapter 7) and linear interpolations (Chapter 10.1). Along the top is a menu (Chapter 13.2) and an independent plane (Chapter 8); both can be controlled with mice (Chapter 12). In the center, the desktop can be seen through the transparent background of the terminal.

2 Right, what's all this, then?

A terminal is at the end of an electric wire, a shell is the home of a turtle, tty is a strange abbreviation and a console is a kind of cabinet.

Gilles Leblanc[76]

Character graphics, aka text mode, aka the display side of a terminal, is visualization that works with fonts rather than a pixel framebuffer¹ or a vector canvas². There is furthermore an expectation that this font is a fixed-width one—that all rendered glyphs are integer multiples of some narrowest non-trivial glyph.

Given the same display hardware,

- Character graphics are usually strictly less powerful than pure raster graphics, and
- their lower effective resolution typically implies lower bandwidth requirements.

A TUI (text user interface) is a holistic model, view, and controller implemented using character graphics. TUIs, like WIMP³ GUIs, freely move the cursor around their rectilinear display, as opposed to line-oriented CLIs and their ineluctable marches through the scrolling region.

Given the same interactive task,⁴

- A TUI implementation is almost certainly a smaller memory and disk footprint than a GUI,
- a good TUI implementation might introduce less latency, and
- a properly-done TUI implementation can often be significantly more portable.

It can also be a big pile of character graphics garbage. A TUI offers less resolution, less flexibility, and (due to monospaced fonts) less total text space. Applications must be carefully designed for the limitations of a dynamic textual environment.

For over two decades, NCURSES (a free software implementation of the X/Open Curses[109] specification, plus extensions[37]) has been a ubiquitous go-to for implementing TUIs. Maintainer (and author, in large part) Thomas E. Dickey exemplifies conservative and fastidious stewardship. Perfectly lovely TUIs can be built using NCURSES (as seen in Figure 3), but it *does* have its origins in the 8-bit era, and shows its age.

Implementing a TUI will usually require, at a minimum:

- Receiving input from user devices, including keyboards and mice,
- some manner of user configuration flow (menus, etc.),
- watching for some other event(s) from the system, and,
- juggling these various components without wastefully polling, nor introducing undue latency, and enforcing safe synchronized access to the graphics interface.

Perhaps most terrifyingly, it will require user interface design. Notcurses attempts to assist with this by providing numerous ready-made widgets.

¹“Contones”, as raster graphics are known to printers.

²Nothing keeps you from implementing character graphics with pixels or vectors, of course.

³Windows, icons, menus, pointers, a paradigm so pervasive that the industry collectively treasures a Wiemarian[101] memory of Xerox PARC's noble engineers stabbed in the back by management (not unlike the Oatmeal-fostered[57] myopia regarding Edison and Tesla. I'll take Thomas Alva over Matthew Inman any day). It too often goes unmentioned that the Alto and Star were as unusable as they were visionary[52]. This is of course still superior to Java, which isn't even visionary.

⁴These relations are not fundamental, but emerge from the grim meathook realities of GUI toolkits.

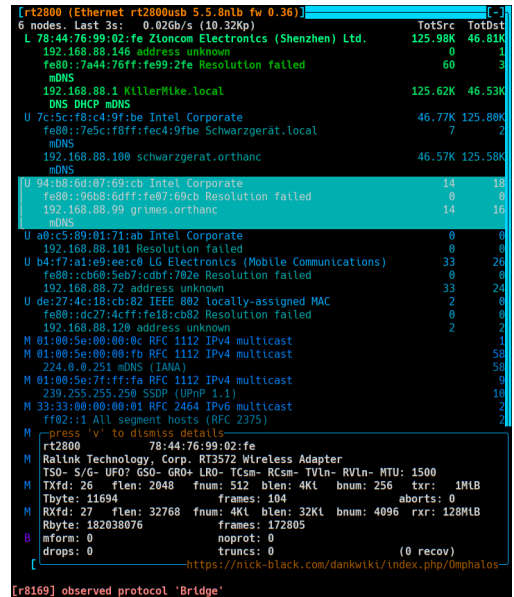
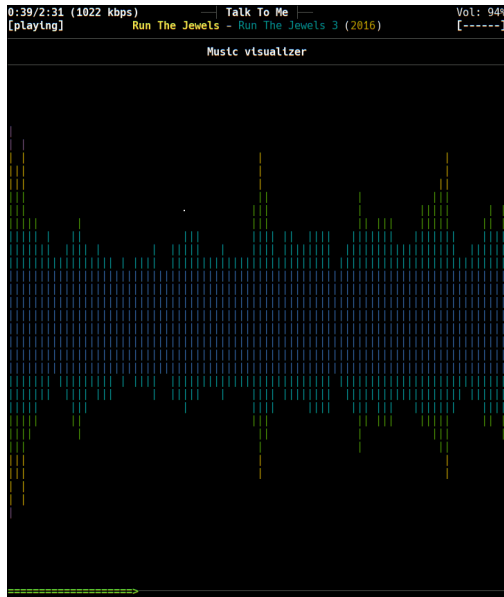


FIGURE 3: Left: `ncmpcpp`, a C++ application that has driven my Music Player Daemon since 2008 or so. Right: `omphalos`, a C network exploration tool written using NCURSES in its extended mode.

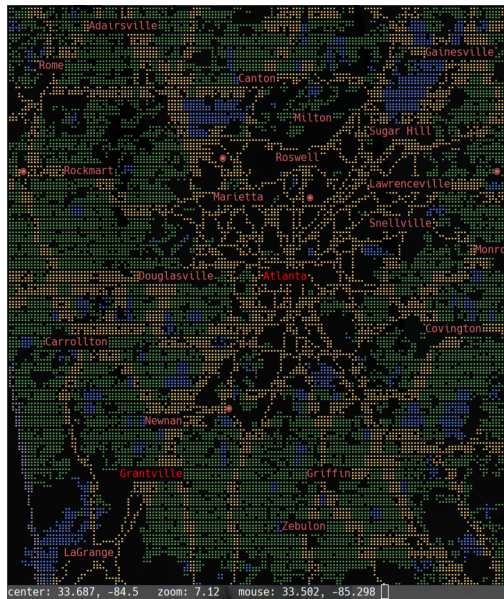


FIGURE 4: Left: `mapscii`, a node.js application, blew my mind when I first saw it. The high resolution is achieved by using Braille characters, trading away some color control. Right: `growlight`, a disk manager, began life as an NCURSES C program, but was ported to Notcurses in 2019.

This text has two goals:

- To provide a firm footing for design and implementation of character graphics and TUIs, elucidating the dimensions of design, along with difficulties to avoid, and
- to serve as “narrative reference” for my Notcurses library, and as a starting place for newcomers.

Cell graphics are primarily the realm of *terminals*, which for the purposes of this book encompass any means by which input devices act to drive some process generating glyph-based output to a display. This includes

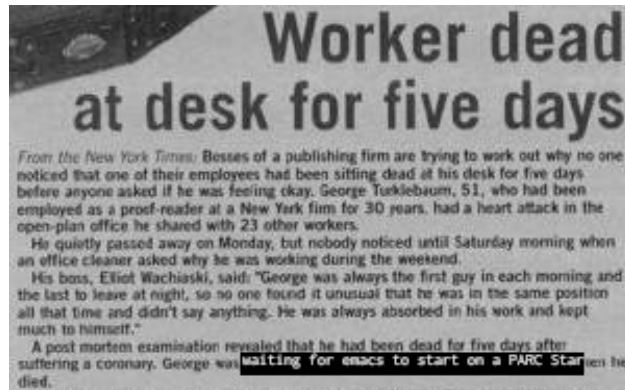


FIGURE 5: Put not your trust in hackers making a fetish of Xerox PARC.

hardware terminals (inputs integrated with displays, connected to a computer as a unit), operating system consoles (text-mode interfaces operating with the graphics engine directly connected to the terminal driver), terminal multiplexers (tools like `screen`, `tmux`, and `mosh`, providing a memory-persistent virtual terminal with which other terminals can interact), and terminal emulators (applications which present a virtual terminal atop the shared input and raster output methods of a graphical user environment). There’s some vagueness and variety involved with these terms.

At its heart, a terminal is a line discipline plus two buffers: an input buffer to collect user-generated events (possibly from multiple devices), and an output buffer to be processed and displayed. The buffers can be modeled as byte streams, mutating the output at the time of their display (in contrast to e.g. a framebuffer, where the entirety of the screen is present at any given time). The earliest terminals were electromechanical teletypes, reproducing their input as line-based print on paper. These gave rise to “dumb terminals” (cathode-ray displays with a scrolling rectilinear output area). “Smart terminals” followed, with the ability to move freely within their display area, and also to extract and act upon “control codes” embedded in the output stream. The text modes of the first video cards were designed around the capabilities of these smart terminals. This brings us to the present, wherein high-powered LED displays have their pixels summoned up and ordered into formations suitable for the reconstruction of 1970s technology (a history of terminals is presented in Appendix A).

The machine on which I’m preparing this $\text{X}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$ contains a TU104 GPU consisting of over thirteen billion 12nm-process transistors, rendering its output to a 3440x1440 (almost five megapixel) display. Deep within its silicon heart remains a VGA 80x25 text mode engine^[92], inherited largely unchanged from the EGA, the CGA⁵, the IBM Monochrome Display Adapter⁶, and before that smart terminals⁷.

I mainly use this modern marvel to drive terminal emulators of 80 columns.

⁵The Color Graphics Adapter was unmitigated trash, but you could do some crazy things with it. People were still finding things out this decade^[69], resulting in the nigh-obsene “8088 MPH” demo that won Revision 2015^[117].

⁶The history of video display standards since 1981’s MDA is a story of imprecision, dashed hopes, and idle dreams. Good luck finding authoritative references for anything beyond int 10h real mode operation prior to version 1.0 of the SuperVGA VESA BIOS Extension^[105], released 1989-10-01^[6].

⁷As early as 1971, the block-oriented IBM 3277 Model 2 “green screen” shipped with 80x24.

3 Using direct mode with standard IO

Unscrew the locks from the doors!
Unscrew the doors themselves from their
jambs!

Walt Whitman, *Song of Myself*

Many tools don't intend to be full-screen TUI applications, but instead implement that purest of UNIX interfaces: newline-delimited text, oblivious to screen geometry, capable of being fed as input to other, similar programs. For such tools, the full Notcurses capabilities are neither necessary nor desirable. These programs are typically non-interactive: humans might peruse their outputs and prepare their inputs, but they effectively run as a batch task.

Such tools might still want to colorize and otherwise style their output, at least when being output to a terminal. This can be accomplished using the `ncdirect` subset of Notcurses, and is known as *direct mode*. Direct mode functionality should not usually be mixed with other Notcurses calls. Unlike full Notcurses, there is no explicit rendering step in direct mode, and it is intended to be mixed among other use of standard I/O. Essentially, direct mode “styles your `printf()`s.” Similarly to full Notcurses, direct mode requires a valid and correct terminfo database entry, supplied via either the `termttype` parameter to `ncdirect_init()` or the `TERM` environment variable. It does *not*, however, require any particular encoding or other locale properties^[100] (full Notcurses requires a properly-configured ASCII or UTF-8 locale).

Enter direct mode via a call to `ncdirect_init()` with a successful return of a non-NULL pointer to `struct ncdirect`. It is typical to invoke this function as `ncdirect_init(NULL, stdout)`. In this case, the terminal type must be present in the `TERM` environment variable (this should have been done by the terminal). The buffering and blocking status of `fp` will not be changed. `NULL` is returned for any number of possible errors. Otherwise, the `struct ncdirect` is ready to go, and should be cleaned up with `ncdirect_stop()`.

```
// Initialize a direct-mode notcurses context on the connected terminal at 'fp'. 'fp' must be a tty. You'll usually
// want stdout. Direct mode supports a limited subset of notcurses routines which directly affect 'fp', and neither
// supports nor requires notcurses_render(). This can be used to add color and styling to text in the standard
// output paradigm. Returns NULL on error, including any failure initializing terminfo.
struct ncdirect* ncdirect_init(const char* termttype, FILE* fp);

// Release 'nc' and any associated resources. 0 on success, non-0 on failure.
int ncdirect_stop(struct ncdirect* nc);
```

LISTING 1: Initializing and stopping direct mode.

Between these two calls, inject stylizing control codes into the `FILE*` with the `ncdirect` (the `stylebits` values are detailed in Chapter 9.2). As detailed in Chapter 9.3, the terminal has a “default foreground color” and “default background color”. Return to these default colors with `ncdirect_fg_default()` and `ncdirect_bg_default()`.

```
int ncdirect_bg_rgb8(struct ncdirect* n, unsigned r, unsigned g, unsigned b);
int ncdirect_fg_rgb8(struct ncdirect* n, unsigned r, unsigned g, unsigned b);
int ncdirect_fg(struct ncdirect* n, unsigned rgb);
int ncdirect_bg(struct ncdirect* n, unsigned rgb);
int ncdirect_styles_set(struct ncdirect* n, unsigned stylebits);
int ncdirect_styles_on(struct ncdirect* n, unsigned stylebits);
int ncdirect_styles_off(struct ncdirect* n, unsigned stylebits);
int ncdirect_clear(struct ncdirect* n);
int ncdirect_fg_default(struct ncdirect* n);
int ncdirect_bg_default(struct ncdirect* n);
```

LISTING 2: The `ncdirect` styling API.

Direct mode provides helpers for determining the terminal geometry.

```
int ncdirect_dim_x(const struct ncdirect* nc);
int ncdirect_dim_y(const struct ncdirect* nc);
```

LISTING 3: Geometry discovery with `ncdirect`.

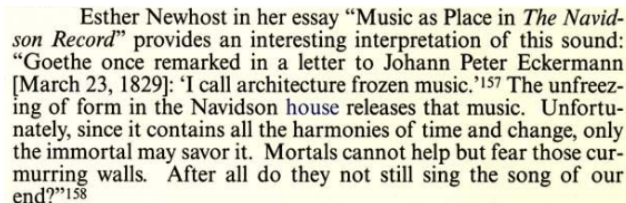
Direct mode allows the cursor to be disabled, enabled, and moved in two-dimensional space. Either `y` or `x` may be specified as `-1` to maintain location on the associated axis.

```
int ncdirect_cursor_move_yx(struct ncdirect* n, int y, int x);
int ncdirect_cursor_enable(struct ncdirect* nc);
int ncdirect_cursor_disable(struct ncdirect* nc);
```

LISTING 4: Cursor management with `ncdirect`.

3.1 Example: presenting *House of Leaves*

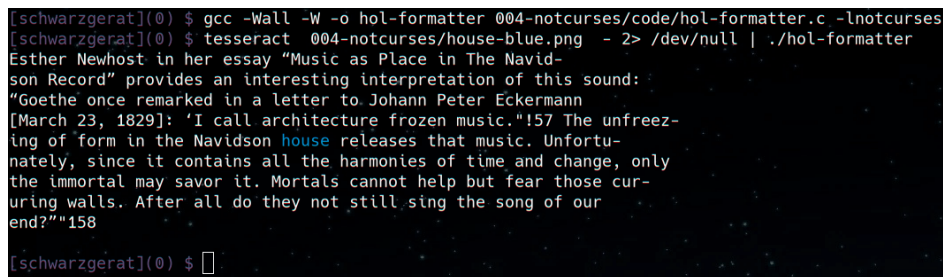
Mark Z. Danielewski’s experimental 2000 novel *House of Leaves*^[34] prints each instance of the word `house` in blue, even when it is a subword:



Esther Newhost in her essay “Music as Place in *The Navidson Record*” provides an interesting interpretation of this sound: “Goethe once remarked in a letter to Johann Peter Eckermann [March 23, 1829]: ‘I call architecture frozen music.’¹⁵⁷ The unfreezing of form in the Navidson **house** releases that music. Unfortunately, since it contains all the harmonies of time and change, only the immortal may savor it. Mortals cannot help but fear those curmurring walls. After all do they not still sing the song of our end?”¹⁵⁸

FIGURE 6: An excerpt from page 123 of *House of Leaves*.

We can easily write code to reproduce this effect for standard input and output. Listing 5 works as expected (see Figure 7), but there are a few things worth noting about its code. First, observe how much of the logic is devoted to checking and propagating errors! Perhaps contrary to common expectation, reliable code—especially when that code’s primary effect is to write to `stdout`—generally needs to check the results of e.g. `printf()` (what happens if we’re redirected to a file, and the disk is full?). A language making use of exceptions would reduce if not eliminate this nonsense.



```
[schwarzgerat]@() $ gcc -Wall -W -o hol-formatter 004-notcurses/code/hol-formatter.c -lnotcurses
[schwarzgerat]@() $ tesseract 004-notcurses/house-blue.png - 2> /dev/null | ./hol-formatter
Esther Newhost in her essay "Music as Place in The Navid-
son Record" provides an interesting interpretation of this sound:
"Goethe once remarked in a letter to Johann Peter Eckermann
[March 23, 1829]: 'I call architecture frozen music.'157 The unfreez-
ing of form in the Navidson house releases that music. Unfortu-
nately, since it contains all the harmonies of time and change, only
the immortal may savor it. Mortals cannot help but fear those cur-
murring walls. After all do they not still sing the song of our
end?"158
[schwarzgerat]@() $
```

FIGURE 7: `hol-formatter` as run on our input. We use `tesseract` for OCR, with solid results.

So long as we’re dealing with either ASCII or UTF-8 input, our simple, old-skool `tolower(3)` is satisfactory *for this problem*. The key observation is that UTF-8 encoded text can be compared for equality by a structure-oblivious `memcmp(3)`, as of course can ASCII. Unless we need to color e.g. `ñöü@ε` (maybe we should, maybe we shouldn’t) this is safe, simple, and sufficient. If we *do* wish to collapse distinct but by some measure similar EGCs, we should normalize input as prescribed by Unicode Standard Annex #15^[24].

```

#include <stdlib.h>
#include <ncurses/direct.h>

int main(void){
    const char blue[] = "house";
    const char *b = blue;
    struct ncdirect* n = ncdirect_core_init(NULL, stdout, 0);
    int c, ret = 0;
    if(n){
        while(!ret && (c = getchar()) != EOF){
            if(isalpha(c) && tolower(c) == *b){
                ++b;
            }else{
                if(b > blue){
                    if(!*b){
                        ret |= ncdirect_on_styles(n, NCSTYLE_BOLD);
                        ret |= ncdirect_set_fg_rgb(n, 0x0339dc);
                    }
                    ret |= (printf("%.*s", (int)(b - blue), blue) < 0);
                    if(!*b){
                        ret |= ncdirect_set_fg_default(n);
                        ret |= ncdirect_off_styles(n, NCSTYLE_BOLD);
                    }
                    b = blue;
                }
                ret |= (putchar(c) == EOF);
            }
        }
        if(b > blue){
            ret |= (printf("%.*s", (int)(b - blue), blue) < 0);
        }
    }
    return (!n || ncdirect_stop(n) || !feof(stdin) || ret) ? EXIT_FAILURE : EXIT_SUCCESS;
}

```

LISTING 5: hol-formatter.c, a streaming formatter.

We don't switch from blue to some other specified color, because we don't know the background color of the terminal. Some people, possibly aliens, don't favor a dark terminal background. If the terminal background were white, and we had just used e.g. `ncdirect_fg(n, 0xffffffff)`, text following “house” would be invisible.

One might observe that a user with a blue background will have invisible “house” text. This is a real issue, one lacking a perfect solution⁸. It is not generally possible to discover the RGB values of the default colors. I suppose all one can do is rest easy, serene in the belief that white backgrounds are one thing, but people with chromatic backgrounds deserve whatever happens to them.

3.2 Example: colorizing a dumb game

Imagine we've written the simple guessing game in Listing 6.

The correct approach for a player is binary search, and for an N -bit long, we expect to guess the number in no more than N tries. Let's color the output to indicate how bad of a guess was offered. We'll use red for low guesses, blue for high guesses, and break the 256 shades of each (assuming the other two components

⁸Applying `NCSTYLE_STANDOUT` might or might not help.

```

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main(void){
    srand(time(NULL)); // gross
    long g, secret = random();
    int r = 0;
    do{
        if(!(r |= printf("Guess the long: ") < 0)){
            if(!fflush(stdout)){
                int rargs = scanf("%ld", &g); // super shitty to the max
                if(rargs != 1){
                    fprintf(stderr, "Die, infidel!\n");
                    return EXIT_FAILURE;
                }
                r |= (printf(g > secret ? "\tLOL jabronies guess %ld. Too high!\n" : g < secret
                    ? "\tSpineless worm! %ld? Too low!\n" : "%ld is right! ", g) < 0);
            }
        }
    }while(g != secret && !r);
    if(r || printf("You enjoy 20/20 vision into the minds of antimen!\n") < 0){
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

```

LISTING 6: hilostdio.c, a simple guessing game.

to be fixed) uniformly across the N levels of logarithmic distance⁹. If we wanted to do this (see Listing 7) without direct use of RGB color, we'd either need accept fewer shades, or be forced to reprogram the palette.

Stepping through the orders of magnitude¹⁰, we get the expected gradient (Figure 8). Were we to actually play, the response would converge to a balanced, strong green as we approached the correct answer.

3.3 Advanced coexistence with stdio

It is most common to initialize Notcurses with `stdout`, whether in direct mode or fullscreen mode. This isn't the only way to operate, though. By opening the `tty` directly using `/dev/tty`, and providing this `FILE*` to Notcurses, a program passing its standard output to another process can make concurrent use of Notcurses on the display, in either direct or fullscreen mode. This is how the `notcurses-pipe` program works¹¹.

For programs that need to write to the terminal, but want to “overlay” some Notcurses, fullscreen mode won't work (though the program could be run in an `nprocess` widget; see Chapter 13). Direct mode, however, is a possibility. I've not yet written the example¹², but it is possible to, for instance, periodically acquire the current cursor position, move elsewhere on the screen, update a HUD, and return to the departure position. Scrolling could be addressed by retaining a copy of any obliterated output. This would suffer a startup period of one screen, during which the area scrolled above the HUD would be cleared. This could be avoided by aligning the HUD with the top of the terminal.

⁹This would be a good place to employ [gamma correction](#).

¹⁰`__builtin_clz1()` is a compiler intrinsic for *count leading zeroes*. Exhaustive methods for fast `clz1` can be found in [118]. Demonstrating that absolute value of the difference of leading zeroes is a lg_2 difference is left as an exercise for the reader.

¹¹See <https://github.com/dankamongmen/notcurses/issues/381>.

¹²Send me patches! Or I'll do it...eventually <https://github.com/dankamongmen/notcurses/issues/382>.

```

#include <limits.h>
#include <ncurses/direct.h>

int main(void){
    srand(time(NULL)); // gross
    long guess, secret = random();
    struct ncdirect* n = ncdirect_core_init(NULL, stdout, NCDIRECT_OPTION_INHIBIT_CBREAK);
    if(n == NULL){
        return EXIT_FAILURE;
    }
    int r = 0;
    do{
        if(!(r |= (ncdirect_set_fg_default(n)))){
            if(!(r |= (printf("Guess the long: ") < 0))){
                if(!(r |= fflush(stdout))){
                    int rargs = scanf("%ld", &guess); // super shitty to the max
                    if(rargs != 1){
                        r = -1;
                        break;
                    }
                    int offoom = labs(__builtin_clz(guess) - __builtin_clz(secret));
                    if(guess > secret){
                        r |= ncdirect_set_fg_rgb8(n, 0x40, 0x80, offoom * 6);
                        r |= (printf("\tLOL jabronies guess %ld. Too high!\n", guess) < 0);
                    }else if(guess < secret){
                        r |= ncdirect_set_fg_rgb8(n, offoom * 6, 0x80, 0x40);
                        r |= (printf("\tSpineless worm! %ld? Too low!\n", guess) < 0);
                    }
                }
            }
        }
    }
    }while(!r && guess != secret);
    if(r || printf("You enjoy 20/20 vision into the minds of antimen!\n") < 0){
        ncdirect_stop(n);
        return EXIT_FAILURE;
    }
    return ncdirect_stop(n) ? EXIT_FAILURE : EXIT_SUCCESS;
}

```

LISTING 7: hilodirect.c, a colored version of the guessing game.

3.4 Use in multithreaded environments

Direct mode calls reduce to a cached terminfo lookup and `fprint(3)` calls on the provided `FILE*`. The former is read-only; all necessary elements are acquired from terminfo at the time of context creation. The latter has the same thread semantics as `fprintf(3)`: while it is *safe* for multiple threads to concurrently print to the same `FILE*`, there are no guarantees of ordering or even atomicity. Given the existence of multibyte UTF-8 output, let alone potentially lengthy escape sequences, it's thus practically necessary that multiple threads working with the same `FILE*` work exclusively.

Multiple threads may freely call read-only functions such as `ncdirect_fg()`.

```

[schwarzgerat](0) $ gcc -Wall -o hilodirect @04-notcurses/code/hilodirect.c -lno
tcurses
[schwarzgerat](0) $ K=1 ; for i in $(seq 1 63) ; do echo $K ; K=$((K*2)) ; done
| ./hilodirect
Guess the long:      Spineless worm! 1? Too low!
Guess the long:      Spineless worm! 2? Too low!
Guess the long:      Spineless worm! 4? Too low!
Guess the long:      Spineless worm! 8? Too low!
Guess the long:      Spineless worm! 16? Too low!
Guess the long:      Spineless worm! 32? Too low!
Guess the long:      Spineless worm! 64? Too low!
Guess the long:      Spineless worm! 128? Too low!
Guess the long:      Spineless worm! 256? Too low!
Guess the long:      Spineless worm! 512? Too low!
Guess the long:      Spineless worm! 1024? Too low!
Guess the long:      Spineless worm! 2048? Too low!
Guess the long:      Spineless worm! 4096? Too low!
Guess the long:      Spineless worm! 8192? Too low!
Guess the long:      Spineless worm! 16384? Too low!
Guess the long:      Spineless worm! 32768? Too low!
Guess the long:      Spineless worm! 65536? Too low!
Guess the long:      Spineless worm! 131072? Too low!
Guess the long:      Spineless worm! 262144? Too low!
Guess the long:      Spineless worm! 524288? Too low!
Guess the long:      Spineless worm! 1048576? Too low!
Guess the long:      Spineless worm! 2097152? Too low!
Guess the long:      Spineless worm! 4194304? Too low!
Guess the long:      Spineless worm! 8388608? Too low!
Guess the long:      Spineless worm! 16777216? Too low!
Guess the long:      Spineless worm! 33554432? Too low!
Guess the long:      Spineless worm! 67108864? Too low!
Guess the long:      Spineless worm! 134217728? Too low!
Guess the long:      Spineless worm! 268435456? Too low!
Guess the long:      LOL jabronies guess 536870912. Too high!
Guess the long:      LOL jabronies guess 1073741824. Too high!
Guess the long:      LOL jabronies guess 2147483648. Too high!
Guess the long:      LOL jabronies guess 4294967296. Too high!
Guess the long:      LOL jabronies guess 8589934592. Too high!
Guess the long:      LOL jabronies guess 17179869184. Too high!
Guess the long:      LOL jabronies guess 34359738368. Too high!
Guess the long:      LOL jabronies guess 68719476736. Too high!
Guess the long:      LOL jabronies guess 137438953472. Too high!
Guess the long:      LOL jabronies guess 274877906944. Too high!
Guess the long:      LOL jabronies guess 549755813888. Too high!
Guess the long:      LOL jabronies guess 1099511627776. Too high!
Guess the long:      LOL jabronies guess 2199023255552. Too high!
Guess the long:      LOL jabronies guess 4398046511104. Too high!
Guess the long:      LOL jabronies guess 8796093022208. Too high!
Guess the long:      LOL jabronies guess 17592186044416. Too high!
Guess the long:      LOL jabronies guess 35184372088832. Too high!
Guess the long:      LOL jabronies guess 70368744177664. Too high!
Guess the long:      LOL jabronies guess 140737488355328. Too high!
Guess the long:      LOL jabronies guess 281474976710656. Too high!
Guess the long:      LOL jabronies guess 562949953421312. Too high!
Guess the long:      LOL jabronies guess 1125899906842624. Too high!
Guess the long:      LOL jabronies guess 2251799813685248. Too high!
Guess the long:      LOL jabronies guess 4503599627370496. Too high!
Guess the long:      LOL jabronies guess 9007199254740992. Too high!
Guess the long:      LOL jabronies guess 18014398509481984. Too high!
Guess the long:      LOL jabronies guess 36028797018963968. Too high!
Guess the long:      LOL jabronies guess 72057594037927936. Too high!
Guess the long:      LOL jabronies guess 144115188075855872. Too high!
Guess the long:      LOL jabronies guess 288230376151711744. Too high!
Guess the long:      LOL jabronies guess 576460752303423488. Too high!
Guess the long:      LOL jabronies guess 1152921504606846976. Too high!
Guess the long:      LOL jabronies guess 2305843009213693952. Too high!
Guess the long:      LOL jabronies guess 4611686018427387904. Too high!
Guess the long:      Die, infidel!
[schwarzgerat](1) $

```

FIGURE 8: Colorized output from hilodirect.c.

4 Using fullscreen mode

This is how space begins, with words only, signs traced on the blank page. To describe space: to name it, to trace it, like those portolano-makers who saturated the coastlines with the names of harbours, the names of capes, the names of inlets, until in the end the land was only separated from the sea by a continuous ribbon of text. Is the aleph, that place in Borges from which the entire world is visible simultaneously, anything other than an alphabet?

Georges Perec, *Species of Spaces*

From this chapter forward, we will be using the fullscreen mode of Notcurses, opening up all of its capabilities. This comes at a cost: while fullscreen mode is being used, it is not safe to use standard I/O in conjunction with the terminal controlled by Notcurses. Doing so is likely to (at a minimum) corrupt the screen. If `stdout` and `stderr` are attached to the same terminal (as they usually are in an interactive session), and `stdout` is provided to Notcurses, output to `stderr` will corrupt the display just as thoroughly as output to `stdout`. If your fullscreen Notcurses program intends to log to `stderr`, you should first ensure that it has been redirected or is otherwise going somewhere different than `stdout`. Note that simply rerendering the output will *not* necessarily clean up corruption, even following `ncplane_erase()` operations, since Notcurses optimizes its rendering based on its concept of the screen. A call to `notcurses_refresh()` will be necessary to sync the physical screen to Notcurses's concept thereof.

It is possible for the screen to be corrupted by external agents. For this reason, `Ctrl+L` is by tradition bound to screen redrawing. You should hook this input up to `notcurses_refresh()` unless you have good reasons not to do so (this is not default behavior of Notcurses only because Notcurses does not itself drive the reading of input). It is sadly not possible for such corruption to be efficiently and generally detected.

It is possible for the attached terminal to be resized, especially (but not only) for terminal emulators in GUI windowing environments¹³. Notcurses can detect such events, and synthesizes `NCKEY_RESIZE` inputs in response to them. If the screen shrinks, the excess data relative to the constant origin will no longer be displayed (i.e. the material in the upper left will be retained). If the screen is enlarged, any data uncovered will be displayed, and the new area will otherwise be empty. Some widgets can intelligently resize themselves in the face of screen geometry changes (see Chapter 13).

Notcurses prepares a given terminal for fullscreen mode in `notcurses_init()`

```
// Initialize a notcurses context on the connected terminal at 'fp'. 'fp' must
// be a tty. You'll usually want stdout. Returns NULL on error, including any
// failure initializing terminfo.
struct notcurses* notcurses_init(const notcurses_options* opts, FILE* fp);

// Destroy a notcurses instance, restoring the terminal to its original state.
int notcurses_stop(struct notcurses* nc);
```

LISTING 8: Initializing and stopping fullscreen mode.

Before calling `notcurses_init()` (and usually as one of the first lines of the program) it is necessary to set the current locale via the standard library function `setlocale()`. A coverage of ANSI/ISO C locales is beyond the scope of this text, but it is usually sufficient to call `setlocale(LC_ALL, "")`, relying on the user's configured `LANG` environment variable. Notcurses only supports those locales using US-ASCII or UTF-8 encodings (see

¹³This could also happen when refitting a screen or `tmux` session. Even on the Linux or FreeBSD console, this can happen due to a change in video resolution.

Chapter 7 for more information on character encodings), and its capabilities on US-ASCII are *severely* constrained. `notcurses_init()` will return an error for any other encoding (see Figure 9).

```
[schwarzgerat](0) $ export LANG="en_GB"
[schwarzgerat](0) $ notcurses-demo
Encoding ("ISO-8859-1") was neither ANSI_X3.4-1968 nor UTF-8, refusing to start
[schwarzgerat](1) $ export LANG="en_US.UTF-8"
[schwarzgerat](0) $ unset TERM
[schwarzgerat](0) $ notcurses-demo
Terminfo error -1 (see terminfo(3ncurses))
[schwarzgerat](1) $ export TERM=xterm-256color
[schwarzgerat](0) $ notcurses-demo
Term: 80x24 xterm-256color (xterm with 256 colors)

notcurses 1.1.8 by nick black et al
24 rows, 80 columns (30.00KiB), 256 colors (direct)
compiled with gcc-9.2.1 20200203
terminfo from ncurses 6.1.20191019
avformat 58.37.100
avutil 56.38.100
swscale 5.6.100

[schwarzgerat](130) $
```

FIGURE 9: Notcurses refusing to start due to an unsupported character encoding.

By default (assuming the `enter_ca_mode` terminfo capability is expressed), Notcurses attempts to enter the “alternate screen”. Using the alternate screen implies:

- The screen will be cleared upon entry,
- Output will not be appended to the scrollbar buffer, and
- On exit, output will be cleared.

Whether or not the original screen contents are restored is terminal-dependent (if the `non_rev_rmcup` terminfo capability is defined, the original contents will *not* be restored). The alternate screen is generally useful, but some users don’t like it, so it’s wise to expose this via a configuration option. Disabling use of the alternate screen can be done via the `notcurses_options` field `inhibit_alternate_screen`.

Successful creation of a struct `notcurses` implies the existence of a struct `ncplane`, the “standard plane”¹⁴. This standard plane cannot be destroyed without destroying the containing Notcurses context, nor can it be moved or resized by the user. Its size always matches Notcurses’s concept of the terminal’s screen size, and its origin always corresponds precisely to the terminal’s origin¹⁵. Aside from these restrictions, the standard plane is a drawable surface like any other `ncplane`—it can be moved along the z-axis, written to with arbitrary glyphs and styles, made transparent, etc.

Once you’re done using a struct `notcurses`, it’s important to destroy it with `notcurses_stop()`, even if your process exits abnormally. By default, Notcurses registers signal handlers for most fatal signals. These handlers will call `notcurses_stop()` and then pass the signal to the original actions. You can disable this with the `no_quit_sighandlers` field of `notcurses_options`, but there aren’t very many good reasons to do so.

4.1 The `notcurses_options` structure

The first parameter to `notcurses_init()` is a (possibly `NULL`) `notcurses_options`. This structure has been defined such that the default options are equivalent to a zero-initialized structure. Passing `NULL` is thus equivalent to passing a zero-initialized `notcurses_options`¹⁶. The fields therein include:

¹⁴`ncplanes`, discussed in depth in Chapter 8, are the fundamental drawing surfaces of Notcurses.

¹⁵Some will note clear similarities to the X “root window”[80].

¹⁶Except it’s strong against changes to `notcurses_options`’s size!

- `const char* termtype`: The name of the terminfo database entry to use. If `NULL`, the value of the environment variable `TERM` is used. Failure to initialize the terminfo database will result in a `notcurses_init()` failure. A defined but invalid or suboptimal entry can result in garbage, missing output, poor performance, reduced colors, and unsightly weight gain.
- `bool inhibit_alternate_screen`: As noted above, this prevents Notcurses from making use of the alternate screen, even if the `enter_ca_mode` terminfo capability is defined. It's best to wire this up to a user-managed option. Not using the alternate screen can look weird upon return to the shell (see Figure 10).

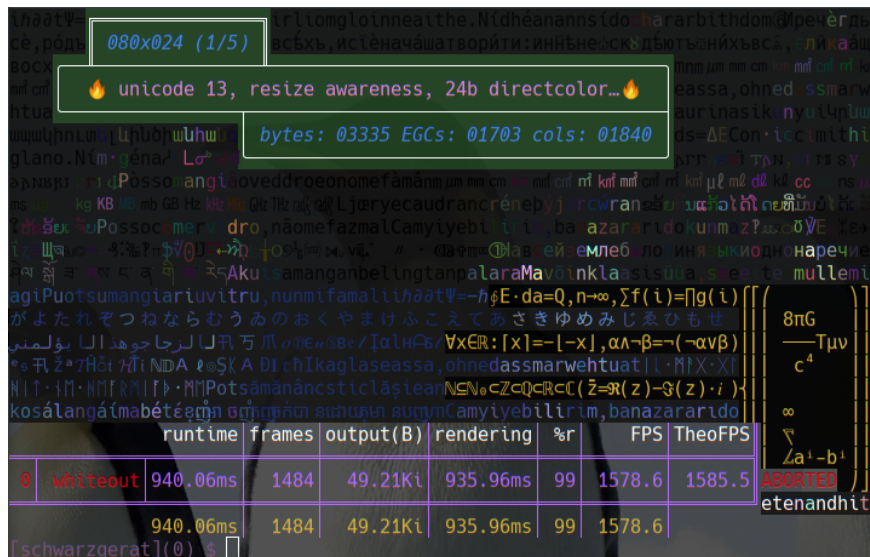


FIGURE 10: `notcurses-demo` can be invoked with `-k` to avoid using the alternate screen. Here, we see its output left on the screen as we return to our shell.

- `bool retain_cursor`: Notcurses hides the cursor by default. Set this to keep the cursor visible (the cursor can be turned on and off at runtime with `notcurses_cursor_enable()` and `notcurses_cursor_disable()`).
- `bool suppress_banner`: At startup, Notcurses emits some diagnostics and/or warnings, including version information and details about the current terminal. At shutdown, it prints performance statistics. These outputs *do not* go to the alternate screen. Set this field to disable these outputs, but be aware that doing so might hide important warnings (see Figure 11).
- `bool no_quit_sighandlers`, `bool no_winch_sighandler`: As noted above, Notcurses by default registers signal actions for the normally fatal `SIGABRT`, `SIGINT`, `SIGQUIT`, and `SIGSEGV`. These handlers will call `notcurses_stop()` before propagating the signal to the original actions. This is usually desirable, as the screen will not otherwise be restored to its previous state. In addition, `SIGWINCH` is caught in order to generate `NCKEY_RESIZE` inputs. If you disable these handlers, you'll almost certainly want to replace them with similar functionality.
- `FILE* renderfp`: If not `NULL`, this designates a file handle open for writing. In addition to the terminal, each rendered scene will be written to this file. This is intended for debugging.
- `int margin_t`, `int margin_r`, `int margin_b`, `int margin_l`: Margin requests on the top, right, bottom, and left, respectively, of the rendering area (see Figure 12). These requests will be satisfied on a best-effort basis—requesting more margin than is actually available is not an error. There must always be at least one row and one column available. If the alternate screen is being used, the margin areas will be cleared. Otherwise, they will be left uncleared. The margins are recomputed on a resize.

```
[schwarzgerat](0) $ ./notcurses-demo -k
Term: 80x26 xterm-256color (xterm with 256 colors)

notcurses 1.1.8 by nick black et al
26 rows, 80 columns (32.50KiB), 256 colors (palette)
compiled with gcc-9.2.1 20200203
terminfo from ncurses 6.1.20191019
avformat 58.37.100
avutil 56.38.100
swscale 5.6.100

Warning! Colors subject to https://github.com/dankamongmen/notcurses/issues/4
Specify a (correct) DirectColor TERM, or COLORTERM.

[schwarzgerat](130) $ export COLORTERM=24bit
[schwarzgerat](0) $ ./notcurses-demo -k
Term: 80x26 xterm-256color (xterm with 256 colors)

notcurses 1.1.8 by nick black et al
26 rows, 80 columns (32.50KiB), 256 colors (direct)
compiled with gcc-9.2.1 20200203
terminfo from ncurses 6.1.20191019
avformat 58.37.100
avutil 56.38.100
swscale 5.6.100
```

FIGURE 11: Initializing Notcurses without 24-bit color support will generate a warning, hopefully provoking your users to set it up.

4.2 Functions on `notcurses` objects

Output is not written to this top-level struct `notcurses`—that’s done with `ncplanes`—but there are a number of functions available for these objects. Acquiring an `ncplane` for output can be done by grabbing a reference to the standard plane, or creating a new plane. New planes are always inserted into the top of the z-axis. All user-created planes can be destroyed in one call with `notcurses_drop_planes()` (note that it is not necessary to call this prior to `notcurses_stop()`; the latter cleans up all resources associated with the context).

Reading input is a per-context operation, performed with `notcurses` objects. It is discussed in detail in Chapter 12. When reading input, we might get the synthesized event `NCKEY_RESIZE`¹⁷. This indicates that the terminal has been resized, and we might want to call `notcurses_resize()` and get the new dimensions. As discussed earlier, sometimes the display is externally corrupted. It’s thus a good idea to hook some UI event (usually `Ctrl+L`) to `notcurses_refresh()`, which redraws every cell on the display according to the internal Notcurses framebuffer.

Finally, `notcurses_render()` synthesizes a terminal’s worth of current state out of all your virtual objects, schedules an optimized list of escape sequences and encoded characters, and blits the result to the terminal. Only through `notcurses_render()` (and transitively through its callers) ought your program write to the actual terminal, and only `notcurses_render()` has any bearing on what the user sees. Between calls, you are free to do whatever you want in terms of moving, reordering, creating, writing upon, and destroying planes. There will be no flicker or tearing; what you last rendered remains on the screen. When you’ve got your stack how you want it, and only then, invoke `notcurses_render()`. It is an exclusive function—any concurrent use of the same struct `notcurses` is undefined.

4.3 Reading, rendering, rasterizing, and writing

Understanding how Notcurses translates its data structures into a terminal display is critical for reasoning about your program in general, and particularly relevant for maximizing performance.

During initialization of a terminal, unless `suppress_banner` is supplied in `notcurses_options`, `notcurses_init()` will print some diagnostics to `stdout`, and flush the output buffer. Notcurses maintains an internal virtual

¹⁷This event is generated upon receipt of a `SIGWINCH` signal, SIGNifying WINdow CHange.


```

// Get a reference to the standard plane (one matching our current idea of the
// terminal size) for this terminal. The standard plane always exists, and its
// origin is always at the uppermost, leftmost cell of the terminal.
struct ncplane* notcurses_stdplane(struct notcurses* nc);
const struct ncplane* notcurses_stdplane_const(const struct notcurses* nc);

// notcurses_stdplane(), plus free bonus dimensions written to non-NULL y/x!
static inline struct ncplane* notcurses_stddim_yx(struct notcurses* nc, int* restrict y, int* restrict x){
    struct ncplane* s = notcurses_stdplane(nc); // can't fail
    ncplane_dim_yx(s, y, x); // accepts NULL
    return s;
}

// Return our current idea of the terminal dimensions in rows and cols.
static inline void notcurses_term_dim_yx(struct notcurses* n, int* restrict rows, int* restrict cols){
    ncplane_dim_yx(notcurses_stdplane(n), rows, cols);
}

// Create a new ncplane at the specified offset (relative to the standard plane)
// and the specified size. The number of rows and columns must both be positive.
// This plane is initially at the top of the z-buffer, as if ncplane_move_top()
// had been called on it. The void* 'opaque' can be retrieved (and reset) later.
struct ncplane* ncplane_new(struct notcurses* nc, int rows, int cols, int yoff, int xoff, void* opaque);

// Return the topmost ncplane, of which there is always at least one.
struct ncplane* notcurses_top(struct notcurses* n);

// Destroy any ncplanes other than the stdplane.
void notcurses_drop_planes(struct notcurses* nc);

// Retrieve the contents of the specified cell as last rendered. The EGC is returned, or NULL on error.
// This EGC must be free()'d by the caller.
char* notcurses_at_yx(struct notcurses* nc, int yoff, int xoff, uint32_t* attr, uint64_t* channels);

```

LISTING 9: Essential functions on notcurses objects.

```

// Refresh our idea of the terminal's dimensions, reshaping the standard plane
// if necessary. References to ncplanes (and the egcpools underlying cells)
// remain valid following a resize, but the cursor might have changed position.
int notcurses_resize(struct notcurses* n, int* restrict y, int* restrict x);

// Refresh the physical screen to match what was last rendered (i.e., without
// reflecting any changes since the last call to notcurses_render()). This is
// primarily useful if the screen is externally corrupted.
int notcurses_refresh(struct notcurses* n);

```

LISTING 10: Dealing with external events.

```

// Make the physical screen match the virtual screen. Changes made to the
// virtual screen (i.e. most other calls) will not be visible until after a
// successful call to notcurses_render().
int notcurses_render(struct notcurses* nc);

```

LISTING 11: Rendering syncs the physical display to our visual planes.

have changed, this saves a tremendous amount of work. On an 80x45 terminal, if only a 10x10 region of cells have changed, we reduce our bandwidth by about 95%¹⁹. These savings are multiplicative:

- Notcurses doesn't have to `write()` the data (memory copy).
- The terminal doesn't have to `read()` the data (memory copy).
- The terminal doesn't need to process the data (assorted work).
- The terminal doesn't need to write to the display (memory copy).

Whether a cell has been updated is decided at rasterization time. Writing to that cell between calls to `notcurses_render()` does not necessarily mean the cell will be considered damaged when it comes time to write. If the cell has been damaged, it will be emitted, and the virtual framebuffer internal to Notcurses will be updated.

Solving for the desired state of the screen is *rendering*, and this is the first step of `notcurses_render()`. Solving for the screen means solving for the current state of every cell, given our ordered set of `ncplanes`. Solving for a cell means determining the extended grapheme cluster to be rendered, determining the attributes to be applied to that EGC, and determining the colors in which it ought be displayed. The higher a plane is on the z-axis, the more it can impact these solutions:

- The EGC and attribute are determined by the first plane intersecting with the cell having a non-null EGC at the intersecting coordinate. If there is no such intersecting EGC, the EGC is null, and the attribute is `NCSTYLE_NORMAL`. Null EGCs are rendered as spaces (i. e. entirely background color).
- The foreground color is determined by the first instance of a `CELL_ALPHA_OPAQUE` foreground color, or an instance of the default foreground color, or an instance of a palette-indexed foreground color, as well as any `CELL_ALPHA_HIGHCONTRAST` or `CELL_ALPHA_BLEND` foreground colors encountered along the way. If there is no such intersecting terminator, the foreground color is the color as calculated thus far. If `CELL_ALPHA_HIGHCONTRAST` is in play, the calculated color is then blended to stand out against the calculated background color.
- The background color is determined independently, in the same way as the foreground color, except without the complicating possibility of `CELL_ALPHA_HIGHCONTRAST`.

Once a cell is solved, Notcurses needn't continue inspecting lower planes at that coordinate. Once all cells are solved, rendering is complete, and any planes left over can be skipped entirely. Until then, Notcurses steps down from one plane to the next, starting at the topmost plane, and updates its solution for any intersecting unsolved cells. It is thus generally more performant to "hide" planes at the bottom of the stack, ideally behind a large opaque plane, rather than moving them beyond the boundaries of the visible window. Likewise, planes ought be no larger than necessary, so that they intersect with the minimum number of cells. Note that there will always be at least one plane interacting with each visible coordinate, due to the properties of the standard plane.

Having rendered the scene, *rasterization* serializes a buffer to write to the terminal, minimizing the amount of data by moving the cursor over undamaged regions. This is the second step of `notcurses_render()`. Writing this data to the terminal as it's generated is a bad idea for several reasons: it can provoke unnecessary context switches, it results in partially-updated displays, and it definitely involves more system calls. Notcurses instead collects it in one or more large allocations.

Proceeding cell-by-cell from the upper left to the lower right, Notcurses compares the rendering solution set to its internal framebuffer. If a given row is entirely undamaged, it can be skipped. Upon discovering the leftmost damage on a row, an absolute cursor update is performed to the damaged cell. At each damaged cell, the EGC will be emitted, along with any necessary styling information. It is only necessary to emit styling escapes when they change, i. e. we can emit multiple EGCs having the same style after only issuing the appropriate escapes once. An RGB change takes about 14 bytes, a palette index change takes about 6, and reverting to the default 2. For single-byte simple (ASCII) EGCs, an RGB foreground and background

¹⁹10x10 is only 2.7% of 80x45, but there is overhead due to moving the cursor to the region, and then positioning the cursor at the end of each line of the region.

represent 2800% overhead per cell! Eliding styling escapes is thus an important secondary optimization (it’s of course most desirable to not update the cell at all). Using the “default color” as only one of the foreground or background requires emitting the `op` escape followed by the appropriate escape for changing the fore- or background (since `op` changes both at once).

Certain EGCs are understood to be all-foreground or all-background. `U+2588 FULL BLOCK` is all foreground. `U+0020 SPACE` is all background. When such characters are used, `notcurses` will emit whichever character requires the fewest total bytes, taking into account both the UTF-8 encoding length and the current color state.

The upshot is that holding styling constant across a horizontal stretch is very desirable if that range’s content is going to be changing. The most pathological input to `Notcurses` is text that changes its foreground and background on a cell-to-cell basis, especially when specified as RGB, that change from render to render. Certain terminal emulators in particular respond to the resulting deluge of RGB escapes very poorly (see Appendix B). As examples, see the `highcontrast` and `grid` demos of `notcurses-demo`—a large `xterm` can be brought to its knees by these routines.

Each subsequent range of undamaged cells on a line can be skipped over with cursor movements, but as the skip length approaches 1, it becomes less and less advantageous to do so. Rendering performance can be very roughly categorized as inversely proportional to the product of:

- color changes across the rendered screen,
- planar depth before an opaque glyph and background are locked in,
- number of UTF-8 bytes comprising the rendered glyphs, and
- screen geometry.

With these buffers in hand, `notcurses_render()` completes its task by writing them to the terminal. This almost certainly means copying them into a kernel buffer from which the terminal will then (following at least one context switch and two system calls) read. Writing does not, then, necessarily mean that the display has actually been updated, or even that the terminal has read the data. If the terminal doesn’t empty the buffer quickly enough, however, you’ll eventually run out of room and block. It is thus critical to understand that `notcurses_render()` **can block for arbitrary amounts of time**²⁰. Furthermore, if the terminal reads two renderings’ worth of output at the same time, it is likely to immediately enter the final state—you must not assume that a successful `notcurses_render()` is necessarily displayed within any arbitrary time, or indeed that it corresponds with any displayed frame.

With those unhappy truths said, modern workstations ought have no problem pushing `notcurses` onto commodity hardware at maximum framerates, with the terminal faithfully reproducing each rendered scene. Even small microcontrollers ought be able to render `notcurses` without user-perceptible latency. On a powerful desktop with non-pathological output, it’s easy to render in excess of ten thousand frames per second, more than an order of magnitude beyond the refresh capabilities of any existing or likely monitor^[10].

4.4 Capabilities

Different terminals expose different capabilities, and different means of engaging them. These differences are encoded in the terminfo database^[77]. `Notcurses` hides the differences where it can, and is built around those capabilities which are most widely supported. Some applications, however, will want to know details of the underlying implementation. For this purpose, the Capabilities API is provided (Listing 12).

4.5 Statistics

`Notcurses` tracks statistics across its operation, and a snapshot can be acquired using the `notcurses_stats()` function (Listing 13). This function cannot fail. Most of the stats can be reset with `notcurses_reset_stats()`. This function resets all cumulative stats, but not those which describe the current state. Timings for

²⁰But see <https://github.com/dankamongmen/notcurses/issues/214>.

```

// Returns a 16-bit bitmask of supported curses-style attributes (NCSTYLE_UNDERLINE, NCSTYLE_BOLD,
// etc.). The attribute is only indicated as supported if the terminal can support it together
// with color. For more information, see the "ncv" capability in terminfo(5).
unsigned notcurses_supported_styles(const struct notcurses* nc);

// Returns the number of simultaneous colors claimed to be supported, or 1 if there is no color support.
// Note that several terminal emulators advertise more colors than they actually support, downsampling internally.
int notcurses_palette_size(const struct notcurses* nc);

// Can we fade? Fading requires either the "rgb" or "ccc" terminfo capability.
bool notcurses_canfade(const struct notcurses* nc);

// Can we set the "hardware" palette? Requires the "ccc" terminfo capability.
bool notcurses_canchangecolor(const struct notcurses* nc);

// Can we load images/videos? This requires being built against FFmpeg.
bool notcurses_canopen(const struct notcurses* nc);

// Get a human-readable string describing the running notcurses version.
const char* notcurses_version(void);

```

LISTING 12: The capabilities API.

renderings are across the breadth of `notcurses_render()`: they include all per-render preprocessing, output generation, and dumping of the output (including any sleeping while blocked on output to the terminal).

Statistics available include:

- `renders`, `failed_renders`: The number of successful and unsuccessful invocations of `notcurses_render()`. Calls to `notcurses_refresh()` do not show up in either of these stats. A render call can fail due to memory pressure, invalid EGCs, or a failure to successfully write to the output terminal.
- `render_bytes`: The number of bytes written in successful renders. Unsuccessful renders do not count towards the total. Dividing `renders` by `render_bytes` yields the average bytes per (successful) render.
- `render_max_bytes`, `render_min_bytes`: The maximum and minimum number of bytes emitted during a successful render.
- `render_ns`, `render_min_ns`, `render_max_ns`: The total, minimum, and maximum number of nanoseconds spent in `notcurses_render()`, whether the calls were successful or not. These timings are acquired using POSIX timers^[16] with the `CLOCK_MONOTONIC`²¹ implementation.
- `cellemissions`, `cellelisions`: The total number of EGCs written to output, and the number that did not need to be written due to being undamaged.
- `fgemissions`, `fgelisions`: Foreground RGB values written to output, and the number elided.
- `bgemissions`, `bgelisions`: Background RGB values written to output, and the number elided.
- `defaultemissions`, `defaultelisions`: `op` escapes issued to set default colors, and the number elided.
- `fbbytes`: The number of bytes devoted to framebuffers.
- `planes`: The current number of planes. Will never drop below 1.

²¹Wouldn't `CLOCK_MONOTONIC_RAW` be superior? It would, where it's available, which isn't everywhere. It's also substantially more expensive than `CLOCK_MONOTONIC` on Linux. Be aware, then, that NTP adjustments and time suspended *do* show up in timings.


```

typedef struct ncstats {
    // purely increasing (cumulative) stats
    uint64_t renders;           // number of successful notcurses_render() runs
    uint64_t failed_renders;    // number of aborted renders, should be 0
    uint64_t render_bytes;      // bytes emitted to ttyfp
    int64_t  render_max_bytes;  // max bytes emitted for a frame
    int64_t  render_min_bytes; // min bytes emitted for a frame
    uint64_t render_ns;        // nanoseconds spent in notcurses_render()
    int64_t  render_max_ns;    // max ns spent in notcurses_render()
    int64_t  render_min_ns;    // min ns spent in successful notcurses_render()
    uint64_t cellelisions;     // cells we elided entirely thanks to damage maps
    uint64_t cellemissions;    // cells we emitted due to inferred damage
    uint64_t fgelisions;       // RGB fg elision count
    uint64_t fgemissions;      // RGB fg emissions
    uint64_t bgelisions;       // RGB bg elision count
    uint64_t bgemissions;      // RGB bg emissions
    uint64_t defaultelisions;  // default color was emitted
    uint64_t defaultemissions; // default color was elided
    // current state -- these can decrease
    uint64_t fbbytes;          // total bytes devoted to all active framebuffer
    unsigned planes;          // number of planes currently in existence
} ncstats;

// Acquire an atomic snapshot of the notcurses object's stats.
void notcurses_stats(struct notcurses* nc, ncstats* stats);

// Reset all cumulative stats (immediate ones, such as fbbytes, are not reset).
void notcurses_reset_stats(struct notcurses* nc, ncstats* stats);

```

LISTING 13: The statistics API.

4.6 Use in multithreaded environments

To facilitate maximum performance, Notcurses does not perform any locking of its own²². All functions are safe for multiple threads to call with regards to system and standard library resources. Things become more complex when multiple threads wish to “write” to Notcurses.

As mentioned above, it is necessary that `notcurses_render()` calls mutually exclude themselves, and also all other functions which mutate the context. This includes all functions which write to a plane, functions which change the ordering of planes (including deletion of a plane), and even statistics functions. With very few exceptions, calling any Notcurses function concurrently with `notcurses_render()` is an error.

Beyond this “big rendering lock”, functions ought not generally be called concurrently on the same `ncplane`, unless all are reading. It is unsafe, for instance, to call `ncplane_putegc()` (a writing function) concurrently with `ncplane_cursor_yx()` (a reading function). It might not be obvious that functions such as `ncplane_at_yx()` which write to a `cell` bound to the plane are writers, but supplying the `cell` might require writing to the plane’s `egpool`, and thus they are writers. In general, a function is only a reader if its `ncplane` argument is `const`.

Concurrent operations on different planes are safe, unless they are changing the ordering along the z-axis. Note that `ncplane_destroy()` updates the z-ordering by virtue of removing an element, and thus must not be called along with e.g. `ncplane_move_top()`.

²²This might change if Notcurses begins to actively acquire input itself, necessary for certain desirable features. In that case, the locking would only be present in the input layer.

5 A simple notcurses render/event loop

I'm not typically a fan of example-based instruction, preferring to build things up from formal axioms. Following chapters will effect such an approach. First, however, let's work a semi-substantial example covering a varied set of Notcurses routines. We're going to create seven planes, one for each kind of tetrimino, and map an image file to the background. We'll add support for switching between the pieces, rotating them, and sliding them around the screen. Finally, we'll deal with collisions, and fluid handling of screen resizings. In the course of doing so, you'll learn several important Notcurses techniques:

- Drawing and rotating these tetriminos will involve colors, gradients, and transparencies. The former are fundamental drawing tools. The latter is all one needs for sprites.
- Mapping the background will involve image decoding, scaling, and blitting.
- We'll cover most of what's worth knowing regarding input.

This example will be picked up and further developed in the Tetris case study of Chapter 14.²³

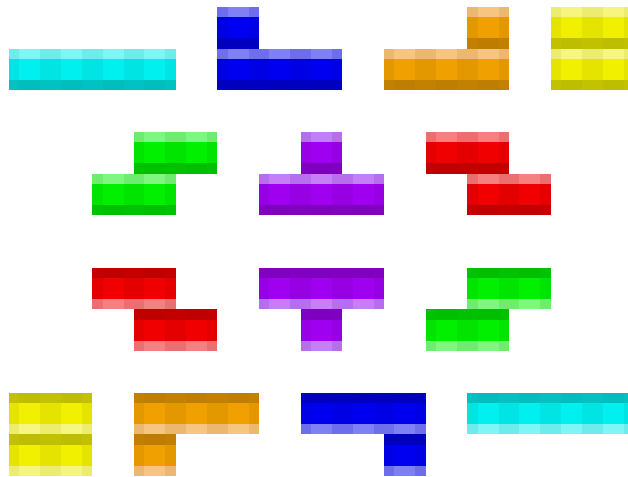


FIGURE 13: Piping hot tetriminos, fresh from *Spiritus Mundi*.

Almost every Notcurses program will take the same general form, an *event+render loop*:

- Essential screen elements are laid down.
- Initial state is discovered and added to the display. **begin loop**
- Some thread, perhaps the only thread in the process, watches for user input. Other threads might be collecting system events that will change the state. Either way, an event occurs.
- Thread(s) receiving events perform any necessary mutual exclusion.
- Thread(s) manipulate the Notcurses virtual state via `ncplane` manipulations.
- `notcurses_render()` is called by a single thread. **end loop**
- The process terminates.

5.1 Example: moving tetriminos with a keyboard

We could load the pieces as images from files, but given the simplicity of the sprites, it feels simpler to just hardcode them in our source. I don't want to leave my text editor²⁴ to muck with images when we can do everything in fewer than ten lines, even in a naïve and wasteful encoding (Listing 14).

²³Were you aware that there is a standard for Tetris clones? There is indeed[56].

²⁴Vim. See Figure 5.

```
// "North-facing" tetrimino forms (the form in which they are released from the top) are expressed in terms of
// two rows having between two and four columns. We map each game column to four columns and each game column
// to two rows. strlen(texture) / COLS_PER_GCOL -> columns per (g)row for the tetrimino.
static const int ROWS_PER_GROW = 2;
static const int COLS_PER_GCOL = 4;

static const struct tetrimino {
    unsigned color;
    const char* texture;
} tetriminos[] = {
    { 0xfecb00, "00000000000000000000000000000000" },
    { 0x09fda, "                IIIIIIIIIIIIIIIIIIIIIIIIIII",
}, { 0x952d98, "    TTTT    TTTT    TTTTTTTTTTTTTTTTTTTTTT",
}, { 0xff7900, "        LLLL    LLLLLLLLLLLLLLLLLLLLLLLLLL",
}, { 0x065bd, "JJJJ    JJJJ    JJJJJJJJJJJJJJJJJJJJJ",
}, { 0x69be28, "    SSSSSSS    SSSSSSSSSSSSSSS    SSSSSSS    ",
}, { 0xed2939, "ZZZZZZZZ    ZZZZZZZZ    ZZZZZZZZ    ZZZZZZZZ" } };

static const size_t TETRIMINO_COUNT = sizeof(tetriminos) / sizeof(*tetriminos);
```

LISTING 14: The seven canonical tetriminos (from tetrimino.c).

To warm up and get limber, let's create seven `nplanes`, one for each type of tetrimino. We'll lay them out in a 2-3-2 formation. This layout will be sloppy, for now: our function `tetrimino_plane()` accepts a piece ID and a coordinate. The plane it creates has its origin at this coordinate. We're not yet adjusting for the size of the planes themselves when coming up with these coordinates—this shifts everything down and to the right from symmetric divisions of the screen, as you'll see.



FIGURE 14: Font aspect ratios center around 0.5.

We use two display rows per game row but four display columns per game column. This reflects what is, at least on my display and font, pretty much a 0.5 aspect ratio (Figure 14).

```

// tidx is an index into tetriminos. yoff and xoff are relative to the
// terminal's origin. returns colored north-facing tetrimino on a plane.
static struct ncplane* tetrimino_plane(struct ncplane* nc, int tidx, int yoff, int xoff){
    const struct tetrimino* t = &tetriminos[tidx];
    const size_t cols = strlen(t->texture) / (2 * ROWS_PER_GROW);
    struct ncplane_options nopts = {
        .y = yoff, .x = xoff, .rows = 2 * ROWS_PER_GROW, .cols = cols,
    };
    struct ncplane* n = ncplane_create(nc, &nopts);
    if(n){
        ncplane_set_fg_rgb(n, t->color);
        size_t y = 0, x = 0;
        for(size_t i = 0; i < strlen(t->texture); ++i){
            if(ncplane_putchar_yx(n, y, x, t->texture[i]) < 0){
                ncplane_destroy(n);
                return NULL;
            }
            y += ((x = (x + 1) % cols) == 0);
        }
    }
    return n;
}

```

LISTING 15: Creating a single tetrimino (from tetrimino.c).

There are a few essential things to take away from Listing 15:

- Each time we call `ncplane_putsimple()`, the cursor is advanced the expected amount. If we were calling `ncplane_putegc()` with a multicolumn grapheme cluster, the cursor would be advanced multiple columns.
- Between rows, we need move the cursor ourselves, since we’re skipping over part of the plane (this isn’t true for several of the pieces, but the cursor update is a trivial operation, not worth avoiding).
- The cursor is always initialized to the origin of a new plane, and coordinates supplied to `ncplane_` functions are relative to the plane, *not* the terminal. Coordinates can be translated among planes using `ncplane_translate()`.
- We only have ASCII characters in the textures right now. Were we to introduce any multibyte UTF-8—and we may well do so, for e. g. the Box Drawing Characters—the `strlen()` we size our rows by here would no longer fly. We’d need use `mbstowcs()`, or redefine the textures as `wchar_t` arrays and use `wcslen()`, or change up our encoding²⁵. *Rien n’est simple, mais tout est facile...*

Our `main()` sets the locale, initializes the terminal, and draws the seven pieces (Listings 16 and 17). Each piece has a one-letter name; for now, we draw with those glyphs. The pieces are monochromatic. This renders familiar shapes in the correct colors (Figure 15), but other than that, it’s (like most classic Curses programs) kinda fugly. I wouldn’t want to play with these pieces.

This is hardly a render/event loop; in fact it’s not a loop at all. If we were using the alternate screen, we wouldn’t even see our output before flashing back to the normal screen. Let’s go ahead and hook up input. We’ll be controlling one of the tetriminos at a time—the selected piece will use boldface, and have its color brightened²⁶. We do so (Listing 18) with a pair of helpers—`reduce()` and `highlight()`—which drive the common `blast()`. `blast()` makes use of two new functions, `ncplane_format()` and `ncplane_stain()`. They allow the attributes

²⁵We go with the third option, as you’ll see.

²⁶Note how we use two distinct indicators. On a monochromatic display, we need the bold, and on a display which can’t do bold, maybe we get a color difference. Also, we want bold even if we have color, because until we see the base color, there’s no reason to think of the initial selection as “bright”.

```

static int draw_tetriminos(struct ncplane* n, struct ncplane** minos, int y, int x){
    if( (minos[0] = tetrimino_plane(n, 0, y / 3, x / 3)) ){
        if( (minos[1] = tetrimino_plane(n, 1, y / 3, x * 2 / 3)) ){
            if( (minos[2] = tetrimino_plane(n, 2, y / 2, x / 4)) ){
                if( (minos[3] = tetrimino_plane(n, 3, y / 2, x / 2)) ){
                    if( (minos[4] = tetrimino_plane(n, 4, y / 2, x * 3 / 4)) ){
                        if( (minos[5] = tetrimino_plane(n, 5, y * 2 / 3, x / 3)) ){
                            if( (minos[6] = tetrimino_plane(n, 6, y * 2 / 3, x * 2 / 3)) ){
                                return 0;
                            }
                            ncplane_destroy(minos[5]);
                        }
                        ncplane_destroy(minos[4]);
                    }
                    ncplane_destroy(minos[3]);
                }
                ncplane_destroy(minos[2]);
            }
            ncplane_destroy(minos[1]);
        }
        ncplane_destroy(minos[0]);
    }
}
return -1;
}

```

LISTING 16: Distributing the tetriminos with “flying-v” technique (from tetrimino.c).

```

int main(void){
    if(!setlocale(LC_ALL, "")){
        return EXIT_FAILURE;
    }
    notcurses_options nopts;
    memset(&nopts, 0, sizeof(nopts));
    nopts.flags = NCOPTION_NO_ALTERNATE_SCREEN;
    struct notcurses* nc;
    if((nc = notcurses_init(&nopts, stdout)) == NULL){
        return EXIT_FAILURE;
    }
    int dimy, dimx;
    struct ncplane* nstd = notcurses_stddim_yx(nc, &dimy, &dimx);
    struct ncplane* minos[TETRIMINO_COUNT]; // 2-3-2
    int failed = draw_tetriminos(nstd, minos, dimy, dimx) || notcurses_render(nc);
    return (notcurses_stop(nc) || failed) ? EXIT_FAILURE : EXIT_SUCCESS;
}

```

LISTING 17: A one-shot, display-only main() (from tetrimino.c).

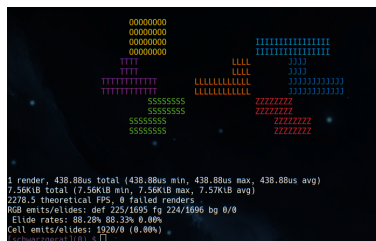


FIGURE 15: Unspeakably foul.

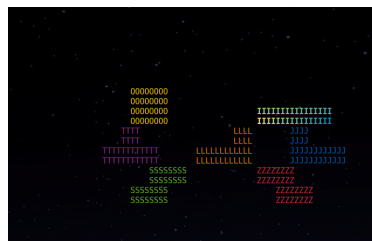


FIGURE 16: Adding a gradient.



FIGURE 17: Linear expansion.

and channels, respectively, of a rectangular region to be changed without altering other components. These, and the corresponding glyph-only output routines, are described in Chapter 10.5.

```
static int blast(struct ncplane* n, uint32_t attrword, uint64_t ul, uint64_t ur, uint64_t ll, uint64_t lr){
    int dimy, dimx;
    ncplane_dim_yx(n, &dimy, &dimx);
    return ncplane_cursor_move_yx(n, 0, 0) ||
           ncplane_format(n, dimy - 1, dimx - 1, attrword) ||
           ncplane_stain(n, dimy - 1, dimx - 1, ul, ur, ll, lr);
}

static int highlight(struct ncplane** minos, int tid){
    int r, g, b;
    uint64_t corig = 0, c = 0;
    r = channel_r(tetriminos[tid].color);
    g = channel_g(tetriminos[tid].color);
    b = channel_b(tetriminos[tid].color);
    channels_set_fg(&corig, tetriminos[tid].color);
    channels_set_fg_rgb_clipped(&c, g + 128, b + 128, r + 128);
    return blast(minos[tid], NCSTYLE_BOLD, corig, c, c, corig);
}

static int reduce(struct ncplane** minos, int tid){
    struct ncplane* n = minos[tid];
    int r, g, b;
    uint64_t c = 0;
    channels_set_fg(&c, tetriminos[tid].color);
    return blast(minos[tid], 0, c, c, c, c);
}
```

LISTING 18: Switching between pieces (from `tetrimino-input.c`).

While we’re making things prettier, let’s replace those letters with some classy box-drawing characters, and improve on this layout. We can do some simple algebraic extensions and get some linear spacing (Figure 17), but it’s just as easy to use trigonometric functions and get an approximation to a circle. This is especially valuable as the terminal geometry changes; our fixed linear scalings would break down as the display aspect changes, but a good ol’ circle will work on any sufficient radius (Listing 19).

Since there’s nothing else going on, it’s trivial to process `stdin` in a blocking fashion from our main thread. Let’s add the code to track a selected piece, visually highlight it, and process input (Listing 20). The space bar will advance among the pieces in one direction, and Tab in the other. The arrow keys and vi keys will translate the selected piece in four directions. The parentheses will rotate the piece $\frac{\pi}{2}$ radians clockwise and counterclockwise.

When there’s no other activity, things are easy. Quite often, we’ll have some periodic concurrent activity. Let’s say we wanted to make the non-selected tetriminos slowly rotate around within their cycle. The rotation isn’t difficult, just movement of the planes. But we don’t want the rotation synced to input activity (the pieces ought rotate freely without the user doing anything), and you’re hardly the kind of know-nothing what-not that would busy loop on a nonblocking input interface, right? I hope you don’t consider yourself “green” if that’s the case, because you’re burning dinosaurs out of sheer laziness. Let’s do it right. There are four canonical solutions to the problem of interleaving a set of asynchronous file descriptor-based inputs against a set of periodic requirements. Let’s refresh ourselves:

- No holds barred, take no prisoners state machine. No quarter asked. None given. Certainly the most *enjoyable* choice of the four, the one closest to the nature of the machine, and the option with the most built-in job security. Casting away the crutch of the process scheduler, we execute in a single

```

static int draw_tetriminos(struct ncplane* nc, struct ncplane** minos, int dimy, int dimx){
    const int centy = dimy / 2, centx = dimx / 2;
    const int radius = dimy < dimx ? dimy / 3 : dimx / 3;
    const float aspect = dimx / (float)dimy;
    const float sector = M_PI * 2 / TETRIMINO_COUNT;
    for(int i = 0 ; i < TETRIMINO_COUNT ; ++i){
        const bool lift = strstrn(tetriminos[i].texture, " ") == strlen(tetriminos[i].texture) / 2;
        if(!minos[i] = tetrimino_plane(nc, i, centy + radius * sin(sector * i) - (ROWS_PER_GROW + lift * 2),
            centx + aspect * radius * cos(sector * i) - strlen(tetriminos[i].texture) / (2 * COLS_PER_GCOL))){
            while(--i){
                ncplane_destroy(minos[i]);
            }
            return -1;
        }
    }
    return 0;
}

```

LISTING 19: Trigonometric layout: simpler, yet more accurate (from `tetrimino-input.c`).

thread, carefully programming our `epoll()`[40] or `kqueue()`[74] timeouts against a hierarchal, hashed timer wheel[114]. As Alan Cox said, “Computers are state machines. Threads are for people who don’t understand state machines.”²⁷ Limited to a single CPU... which probably isn’t a problem here, but isn’t exactly a great foundation on which to build our cathedral, *n’est-ce pas* ?.

- Two threads enter, one thread `exit()`s (hehehehe). We block on async I/O in our main thread. Another thread runs a timer wheel—for this, a loop around a `clock_nanosleep()` will suffice. They lock against one another to avoid corrupting the screen or otherwise embarrassing ourselves. The solution reeks of Java, but will work well enough, and any programmer—maybe even a Java programmer—ought be able to walk in and pick it up.
- POSIX signaled timers. `Nooo`.
- Timer I/O multiplexing. Probably the best solution for a program of true scope, and in no way unfit for the task at hand. On Linux, you’ll want `timerfd_create()` and friends. On FreeBSD, look for `EVFILT_TIMER`. Throw that into the appropriate I/O multiplexor call, spin it out among a few threads if you feel so inclined[11], and call it a day. The only real disadvantage here is a lack of portability.

Some will ask, “Sounds dank, but what about on this shiny operating system I purchased from Applooglesoft? Look, I can speak to it! Sirana, diminish my freedoms and spy on me!” I could give two shits about your closed-source operating system. I presume you can purchase cloud-based Timers as a Service (TaaS) or something.

We can now cycle through the pieces, and move the piece we’ve selected around on the screen. Grab one and move it towards another piece. Experimentation will reveal that each piece has a **bounding box**, that there is a total ordering among the seven pieces, and that a piece below another piece’s bounding box is robbed of its color (Figure 22). Recall from Chapter 4.3 how we solve for each coordinate of the display grid: the EGC is a dimension distinct from the coloring channels. Each of the planes we create is rectilinear, with the piece drawn using Unicode `U+2588 FULL BLOCK`, and the other cells left unwritten. The foreground for our pieces is an RGB color, and for the other cells is the default terminal color. Upon intersecting with a lower plane, the lower plane’s EGC is chosen for rendering, but the foreground color is default terminal color (white in my example). So the piece underneath flipping to white while another piece is nearby makes perfect sense. The solution is simple: we set the foreground transparent for the base character of each plane. Foreground calculation will now bypass the plane where we haven’t drawn, and the distorted plane regains its expected colors (Figure 23). Takeaway: by default, planes obstruct only color, not glyphs.

²⁷Or did he? I can’t find the original to cite. I wonder what the hell happened to Alan Cox.

```

#include <uchar.h>

static bool handle_input(struct notcurses* nc, struct tetmarsh* marsh, int dimy, int dimx, int* y, int* x){
    ncinput ni; // necessary for mouse
    pthread_mutex_unlock(&marsh->lock);
    char32_t key = notcurses_getc_blocking(nc, &ni);
    pthread_mutex_lock(&marsh->lock);
    struct ncplane** ps = marsh->minos;
    const int p = marsh->p;
    if(key == (char32_t)-1){
        return true;
    }else if(key == 'q'){
        marsh->p = -1;
    }else if(key == ' ' || key == '\t'){ // select previous/next
        reduce(ps, p);
        marsh->p = (p + (key == ' ' ? (TETRIMINO_COUNT - 1) : 1)) % TETRIMINO_COUNT;
        ncplane_yx(ps[p], y, x);
    }else if(key == '('){ // rotate counterclockwise
        return ncplane_rotate_ccw(ps[p]);
    }else if(key == ')'){ // rotate clockwise
        return ncplane_rotate_cw(ps[p]);
    }else if(key == NCKEY_LEFT || key == 'h'){
        return ncplane_move_yx(ps[p], *y, --*x < 0 ? *x = 0 : *x);
    }else if(key == NCKEY_RIGHT || key == 'l'){
        return ncplane_move_yx(ps[p], *y, *x = (*x + ncplane_dim_x(ps[p]) > dimx ? *x : *x + 1));
    }else if(key == NCKEY_UP || key == 'k'){
        return ncplane_move_yx(ps[p], --*y < 0 ? *y = 0 : *y, *x);
    }else if(key == NCKEY_DOWN || key == 'j'){
        return ncplane_move_yx(ps[p], ++*y + 2 > dimy ? *y = dimy - 2 : *y, *x);
    }else if(nckey_mouse_p(key)){
        return handle_mouse_event(nc, &ni);
    }
}
return false;
}

```

LISTING 20: Core input dispatch (from tetrimino-input.c).



FIGURE 18: Trigonometry!

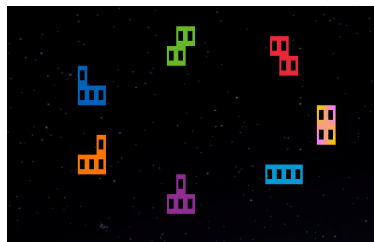


FIGURE 19: Unicode Blocks.

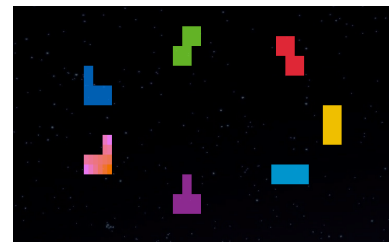


FIGURE 20: Better blocks.



FIGURE 21: Adjusting for cell aspect ratio.



FIGURE 22: Undesirable plane interaction.

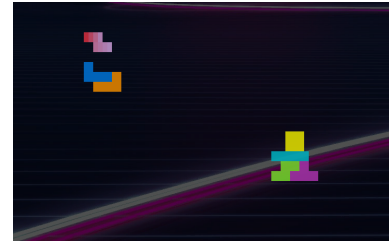


FIGURE 23: Resolve it with transparent planes.

```
// "North-facing" tetrimino forms (the form in which they are released from the
// top) are expressed in terms of two rows having between two and four columns.
// We map each game column to four columns and each game row to two rows.
// Each byte of the texture maps to one 4x4 component block (and wastes 7 bits).
static const struct tetrimino {
    unsigned color;
    const char* texture;
} tetriminos[] = { // OITLJSZ
    { 0xc9900, "****"}, { 0x009caa, "   ****"}, { 0x952d98, " * ***"},
    { 0xcf7900, "  ****"}, { 0x0065bd, "*  ***"}, { 0x69be28, " **** "},
    { 0xbd2939, "** * **"} };

// tidx is an index into tetriminos. yoff and xoff are relative to the
// terminal's origin. returns colored north-facing tetrimino on a plane.
static struct ncplane* tetrimino_plane(struct ncplane* nc, int tidx, int yoff, int xoff){
    const struct tetrimino* t = &tetriminos[tidx];
    const size_t cols = strlen(t->texture);
    struct ncplane_options nopts = {
        .y = yoff, .x = xoff, .rows = ROWS_PER_GROW, .cols = cols,
    };
    struct ncplane* n = ncplane_create(nc, &nopts);
    if(n){
        uint64_t channels = 0;
        ncchannels_set_bg_alpha(&channels, NCALPHA_TRANSPARENT);
        ncchannels_set_fg_alpha(&channels, NCALPHA_TRANSPARENT);
        ncplane_set_fg_rgb(n, t->color);
        ncplane_set_base(n, "", 0, channels);
        size_t y = 0, x = 0;
        for(size_t i = 0 ; i < strlen(t->texture) ; ++i){
            if(t->texture[i] == '*'){
                if(ncplane_putstr_yx(n, y, x, "█") < 0){
                    ncplane_destroy(n);
                    return NULL;
                }
            }
            x += ((x + 2) % cols) == 0;
            y += ((x + 2) % cols) == 0;
        }
    }
    return n;
}
```

LISTING 21: Improving appearance with Unicode Block Elements (from tetrimino-input.c).

Adding a background image is simple. We'll render it to the standard plane, our "background" for now (remember, newly created planes are placed at the top of the z axis). Despite decoding the image *after* creation of the pieces, the pieces are thus not hidden. There is no need to inform Notcurses of the image's format, nor parameters thereof. Simply provide the file name and target rendering plane, and we're off. See Chapter 11 for a full treatment of multimedia in Notcurses.

```
static struct ncvisual* background(struct notcurses* nc, const char* fname){
    struct ncvisual* ncv = ncvisual_from_file(fname);
    if(!ncv){
        return NULL;
    }
    if(ncvisual_render(nc, ncv, NULL) <= 0){
        ncvisual_destroy(ncv);
        return NULL;
    }
    ncplane_greyscale(notcurses_stdplane(nc));
    return ncv;
}
```

LISTING 22: Throwing in a background (from `tetrimino-input.c`).

To rotate the unselected pieces, we'll go ahead and spawn a POSIX thread. We'll thus need lock against piece selection. We've been leaving all the end-of-process cleanup in the capable hands of `notcurses_stop()`, but it can't go terminating threads for us, and it in any case wouldn't be safe to do so while said thread was calling into Notcurses! First, add a marshaling struct to share some state (Listing 23). We'll use POSIX cancellation²⁸ to blast the rotator thread, and `pthread_join()` it to ensure safe passage through shutdown.

```
struct tetmarsh {
    struct ncplane* minos[TETRIMINO_COUNT];
    struct ncplane* coaster;
    pthread_mutex_t lock;
    struct notcurses* nc;
    int p;
};
```

LISTING 23: Marshaling structure for shared state (from `tetrimino-input.c`).

As a final touch, let's slide a dark, translucent plane underneath the selected piece, making it more visible. We create this plane prior to the pieces, ensuring it's below all of them (but above the background). We set it black, and its alpha to `CELL_ALPHA_BLEND`. This way it won't entirely block out the background underneath the selected piece, but it will dim it, its black blending into the grey tones underneath. The piece is unaffected.

²⁸Which isn't anywhere near as bad as it's sometimes made out to be. Find the places where you don't want to be cancelled. Each such place needs a cleanup handler pushed/popped around it, or cancellation disabled for its breadth. Things get a little counterintuitive with cancellable-but-uninterruptible system calls, but that's nothing if you came up on BSD signals. It's hardly the most offensive wart on the great hairy ass that is UNIX systems programming, and—heresy, I know—I honestly prefer (most of) the POSIX model to (most of) the threading introduced in C++ 11.

```

static void unlock_mutex(void* vlock){ pthread_mutex_unlock(vlock); }

// do a 1/2Hz rotation through the circle
static void* rotator_thread(void* vmarsh){
    struct tetmarsh* marsh = vmarsh;
    const float sector = M_PI * 2 / TETRIMINO_COUNT;
    const float ssector = M_PI * 2 / 18; // 20deg movements, so 18 sec/s
    const uint64_t NSTOS = 1000000000ull / 9;
    struct timespec stime;
    clock_gettime(CLOCK_MONOTONIC, &stime);
    uint32_t iterations = 0;
    while(++iterations, true){
        int dimy, dimx;
        struct ncplane* stdn = notcurses_stddim_yx(marsh->nc, &dimy, &dimx); // handles resize
        const float aspect = dimx / (float)dimy;
        int centy = dimy / 2;
        int centx = dimx / 2;
        struct timespec dline = { .tv_sec = 0, .tv_nsec = NSTOS, }; // FIXME abstime with catchup
        while(clock_nanosleep(CLOCK_MONOTONIC, 0, &dline, NULL) < 0){
            if(errno != EINTR){
                return NULL;
            }
        }
        pthread_cleanup_push(unlock_mutex, &marsh->lock);
        pthread_mutex_lock(&marsh->lock);
        const int radius = dimy < dimx ? dimy / 3 : dimx / 3;
        for(int i = 0 ; i < TETRIMINO_COUNT ; ++i){
            const size_t len = strlen(tetriminos[i].texture);
            const bool lift = strstrn(tetriminos[i].texture, " ") == len / 2;
            int y = centy + radius * sin(sector * i + ssector * iterations) - (ROWS_PER_GROW + lift * 2);
            int x = centx + aspect * radius * cos(sector * i + ssector * iterations) - len / (2 * COLS_PER_GCOL);
            if(ncplane_move_yx(marsh->minos[i], y, x)){
                pthread_mutex_unlock(&marsh->lock);
                return NULL;
            }
        }
        int ny, nx;
        ncplane_yx(marsh->minos[marsh->p], &ny, &nx);
        ncplane_move_yx(marsh->coaster, ny - 2, nx - 2 * COLS_PER_GCOL);
        notcurses_render(marsh->nc);
        pthread_cleanup_pop(1);
    }
}

```

LISTING 24: Spin them doggies (from tetrimino-input.c).

```

// makes an opaque box to highlight the selected piece
static struct ncplane* makebox(struct ncplane* nc){
    struct ncplane_options nopts = {
        .rows = 2 * ROWS_PER_GROW + 2, .cols = 4 * COLS_PER_GCOL + 4,
    };
    struct ncplane* ret = ncplane_create(nc, &nopts);
    if(ret){
        uint64_t tl = 0, br = 0, m = 0;
        ncchannels_set_fg_rgb(&tl, 0xffffffff); ncchannels_set_fg_rgb(&br, 0xffffffff); ncchannels_set_fg_rgb(&m, 0xffffffff);
        ncchannels_set_bg_rgb(&tl, 0x000000); ncchannels_set_bg_rgb(&br, 0x000000); ncchannels_set_bg_rgb(&m, 0x000000);
        if(ncplane_gradient_sized(ret, " ", 0, tl, m, m, br, 2 * ROWS_PER_GROW + 2, 4 * COLS_PER_GCOL + 4) >= 0){
            #define CTI(cname) nccell cname = CELL_TRIVIAL_INITIALIZER
            CTI(cul); CTI(cur); CTI(cbl); CTI(cbr); CTI(chl); CTI(cvl);
            #undef CTI
            if(nccells_double_box(ret, 0, br, &cul, &cur, &cbl, &cbr, &chl, &cvl) == 0){
                int r = ncplane_perimeter(ret, &cul, &cur, &cbl, &cbr, &chl, &cvl, 0);
                if(r == 0){
                    ncplane_cursor_move_yx(ret, 0, 0);
                    return ret;
                }
                nccell_release(ret, &cul); nccell_release(ret, &cur); nccell_release(ret, &chl);
                nccell_release(ret, &cbl); nccell_release(ret, &cbr); nccell_release(ret, &cvl);
            }
        }
        ncplane_destroy(ret);
    }
    return NULL;
}

static int highlight_enbox(struct ncplane** minos, int tid, struct ncplane* box){
    int r, g, b, ny, nx;
    uint64_t corig = 0, c = 0;
    r = ncchannel_r(tetriminos[tid].color);
    g = ncchannel_g(tetriminos[tid].color);
    b = ncchannel_b(tetriminos[tid].color);
    ncchannels_set_fg_rgb8_clipped(&c, g + 128, b + 128, r + 128);
    ncplane_yx(minos[tid], &ny, &nx);
    ncplane_move_yx(box, ny - 2, nx - 2 * COLS_PER_GCOL);
    return blast(minos[tid], NCSTYLE_BOLD, c, c, c, c);
}

```

LISTING 25: Set the selection off with a coaster (from tetrimino-input.c).

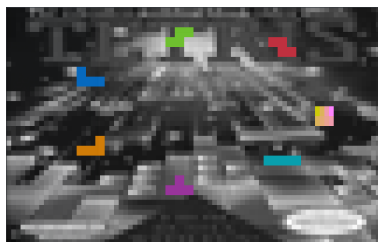


FIGURE 24: Background image, greyscaled.

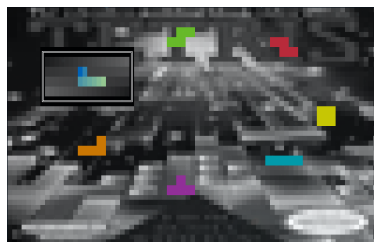


FIGURE 25: Opaque highlight box.



FIGURE 26: Rotation thread enabled.

```

int main(void){
    if(!setlocale(LC_ALL, "")){
        return EXIT_FAILURE;
    }
    struct notcurses* nc;
    if((nc = notcurses_init(NULL, stdout)) == NULL){
        return EXIT_FAILURE;
    }
    int dimy, dimx, y, x;
    struct tetmarsh marsh = { .lock = PTHREAD_MUTEX_INITIALIZER, .nc = nc, .p = 0, };
    struct ncplane* nstd = notcurses_stddim_yx(nc, &dimy, &dimx);
    if(!(marsh.coaster = makebox(nstd)) || draw_tetriminos(nstd, marsh.minos, dimy, dimx)){
        notcurses_stop(nc);
        return EXIT_FAILURE;
    }
    ncplane_yx(marsh.minos[marsh.p], &y, &x);
    int failed = notcurses_render(nc);
    struct ncvisual* ncv = background(nc, "media/Tetris_NES_cover_art.jpg");
    pthread_t tid;
    if(!failed && (failed = pthread_create(&tid, NULL, rotator_thread, &marsh)) == 0){
        pthread_mutex_lock(&marsh.lock);
        while((failed != (highlight_enbox(marsh.minos, marsh.p, marsh.coaster) || notcurses_render(nc))) == 0){
            failed != handle_input(nc, &marsh, dimy, dimx, &y, &x); // emerge locked
            if(marsh.p < 0){
                break;
            }
        }
        pthread_mutex_unlock(&marsh.lock);
        void* result;
        failed != (pthread_cancel(tid) || pthread_join(tid, &result) || result != PTHREAD_CANCELED);
    }
    return (notcurses_stop(nc) || pthread_mutex_destroy(&marsh.lock) || failed) ? EXIT_FAILURE : EXIT_SUCCESS;
}

```

LISTING 26: Putting it all together (from tetrimino-input.c).

6 Terminal mechanics

The tty layer is one of the very few pieces of kernel code that scares the hell out of me.

Ingo Molnar[83]

You won't often need to deal with the gritty details of terminal access and manipulation. It's still important to understand what's going on behind the abstraction, especially for when things go wrong. As was made clear in Chapters 3 and 4, Notcurses requires a proper terminal definition and a handle to a terminal device, or initialization will fail (NCURSES requires the same). With that said, the UNIX terminal layers have never been, and are not now for the faint of heart²⁹. Extending back to the AT&T dark ages (and with at least three major distinct interfaces over the years, only unified in 1997's SUS2), they are configured primarily through messy `ioctl`s and the slightly-less-messy `termios` API.

More details than you probably want are available from Chapters 18 and 19 of [107] (general UNIX), Chapters 62 and 64 of [72] and Chapter 18 of [28] (Linux), and Chapter 10 of [82] (FreeBSD).

Modern workstations support a variety of physical and virtual terminal devices:

- Honest-to-Bog serial terminals, probably using the RS-232[60] protocol over a D-subminiature 25-pin (DB-25M) or 9-pin (DE-9M) connector (see Table 1).
- Virtual consoles on text-based video, plus a keyboard.
- Virtual framebuffer consoles on graphics-based video, plus a keyboard.
- Terminal emulators in a graphical environment, plus brokered input devices.
- Pseudoterminals hooked up to network connections.

Signal	DB-25M	TIA-574 DE-9M	Yost 8P8C[123]	Originator
Protective ground	1	x	x	x
Transmitted data	2	3	3	DTE
Received data	3	2	6	DCE
Request to send	4	7	1	DTE
Clear to send	5	8	8	DCE
Data set ready	6	6	2	DCE
Signal ground	7	5	4, 5	x
Carrier detect	8	1	2	DCE
Data terminal ready	20	4	7	DTE
Ring indicator	22	9	x	DCE

TABLE 1: RS-232/EIA-232 pins (DCE=Data circuit equipment, DTE=Data terminal equipment)

On Linux, three kernel definitions form the core of the terminal abstraction (the “tty layer”). A `tty_driver` function interface exists for each tty implementation, and each has a different corresponding major+minor device number pair. These implementations are enumerated in `/proc/tty/drivers` (sample contents are listed in Table 2). Line disciplines can be found in `/proc/tty/ldiscs`. The only line discipline you're likely to encounter in a terminal context is `n_tty`, the “new tty” discipline responsible for implementing “cooked mode”[116].

POSIX.1-2017[55] §10.1 “Directory Structure and Devices” specifies three special files in the `/dev` directory, of which two are related to the terminal/console system: `/dev/tty` is a synonym for the controlling terminal associated with that process's group (this can also be acquired with `tty(1)` or the POSIX C function `ctermid(3)`³⁰; the tty associated with an arbitrary file descriptor can be retrieved with `ttynam_r(3)`). `/dev/console` is a generic

²⁹When Ingo Molnar is scared, we all ought be scared.

³⁰`/dev/tty` also uniquely supports the `TIOCNOTTY ioctl(2)` for detaching a process from its controlling terminal.

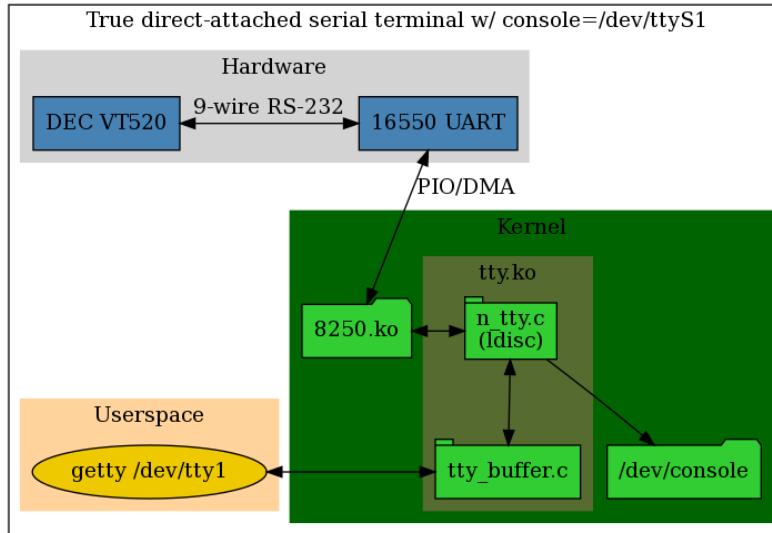


FIGURE 27: A serial console, from hardware to userspace. Adapted from [2].

name for the current system console³¹; POSIX requires that the system console implement its “General Terminal Interface” (see *ibid.* §11). The standard console will by default be `/dev/tty0`, but this can be changed with the `console=` kernel command line parameter, or with an explicit `conOut` UEFI variable (see §11.4 of [43]). `/dev/tty0` is a special device on Linux linked to the current virtual console (which might not be the system console). The available system consoles can be found in `/proc/consoles`.

Initial `/dev/ttyN` devices are created internally by the kernel (according to the value of `MAX_NR_CONSOLES`, by default 64) and set up by `udev`. These are distinct devices, each usable by one virtual console. Serial consoles show up as `/dev/ttySN`[112]. Each virtual console gets a device at `/dev/vcsn` allowing access to the glyph values of the console, a device at `/dev/vcsa` allowing access to the attributes at each cell, plus screen geometry, and a device at `/dev/vcsu` providing access to the Unicode values of each cell[115]. If a framebuffer console is being used, these devices are prepared atop some framebuffer device `/dev/fbN` (see Figure 28); these mappings can be managed with `con2fbmap(1)`.

<pre>[schwarzgerat](0) \$ fbset -l mode "1024x768" geometry 1024 768 1024 768 32 timings 0 0 0 0 0 0 rgba 8/16,8/8,8/0,8/24 endmode Frame buffer device information: Name : simple Address : 0xc0000000 Size : 3145728 Type : PACKED PIXELS Visual : TRUECOLOR XPanStep : 0 YPanStep : 0 YWrapStep : 0 LineLength : 4096 Accelerator : No [schwarzgerat](0) \$</pre>	<pre>[killermike](0) \$ fbset -l mode "1920x1080" geometry 1920 1080 1920 1080 32 timings 0 0 0 0 0 0 accel true rgba 8/16,8/8,8/0,0/0 endmode Frame buffer device information: Name : amdgpudrmfb Address : 0 Size : 8294400 Type : PACKED PIXELS Visual : TRUECOLOR XPanStep : 1 YPanStep : 1 YWrapStep : 0 LineLength : 7680 Accelerator : No [killermike](0) \$</pre>	<pre>[mocha](0) \$ sudo fbset -l mode "2560x1440" geometry 2560 1440 2560 1440 32 timings 0 0 0 0 0 0 accel true rgba 8/16,8/8,8/0,0/0 endmode Frame buffer device information: Name : i915drmfb Address : 0 Size : 14745600 Type : PACKED PIXELS Visual : TRUECOLOR XPanStep : 1 YPanStep : 1 YWrapStep : 0 LineLength : 10240 Accelerator : No [mocha](0) \$</pre>
--	---	--

FIGURE 28: Three different Linux framebuffer implementations.

The various terminos flags allow very fine control of the kernel state associated with a given terminal. It is possible to mix flag settings arbitrarily, but three modes are common, and have their own nomenclature:

³¹Defined in *ibid.* §3.392 “System Console” as the device receiving messages sent by the `syslog()` function. On Linux, it will also reproduce messages written to `/dev/kmsg`[39].

Name	Default node	Major	Minor	Type	sysfs
/dev/tty	/dev/tty	5	0	system:/dev/tty	devices/virtual/tty/tty*
/dev/console	/dev/console	5	1	system:console	devices/virtual/tty/console
/dev/ptmx	/dev/ptmx	5	2	system	devices/virtual/tty/ptmx
/dev/vc/0	/dev/vc/0	4	0	system:vtmaster	x
usbserial	/dev/ttyUSB	188	0-511	serial	x
serial	/dev/ttyS	4	64-95	serial	x
rfcomm	/dev/rfcomm	216	0-255	serial	x
pty_slave	/dev/pts	136	0-1048575	pty:slave	x
pty_master	/dev/ptm	128	0-1048575	pty:master	x
unknown	/dev/tty	4	1-63	console	devices/virtual/tty/console

TABLE 2: Extended content of a sample `/proc/tty/drivers` (5.5.6 kernel. `sysfs` information has been added)

- **Canonical mode, aka cooked mode:** The terminal driver buffers input until a newline is entered, while echoing it to the screen. `Ctrl+C` is mapped to `SIGINT`, `Ctrl+\` is mapped to `SIGQUIT`, and `Ctrl+Z` is mapped to `SIGTSTP`. Buffered input is flushed when these signals are sent. The default mode.
- **Cbreak mode:** Line buffering is disabled, as is the processing of erase/kill characters. Interrupt and flow control translation is unaffected. Sometimes referred to as *rare mode*. This mode allows processing of input without waiting for a newline character, while retaining e.g. `Ctrl+C`.
- **Raw mode:** Interrupt and flow control translation is also disabled. This allows all keyboard input to be processed without preprocessing by the line discipline, but e.g. `Ctrl+C` behavior is lost, and (if desired) must be emulated in user space.

In addition to these physically-backed devices, `ptys`—pseudoterminal devices^[96]—exist to support virtual teletype functionality, wherein a process plays the role of hardware. Pseudoterminals³² back most terminal instances, including GUI terminal emulators, network services such as SSH, and multiplexers such as `screen`. A `pty` provides a bidirectional channel between a *master* and *slave*, with the slave presenting a classic terminal interface (i.e. writing a `Ctrl+C` to the master side will (under the cooked or `cbreak` disciplines) result in `SIGINT` being delivered to the foreground process group).

The details of Linux pseudoterminal pair creation can be found on the `pts(4)`^[95] man page. `posix_openpt(3)`^[89] opens the “master clone device” `/dev/ptmx`, receiving a file descriptor corresponding to the new PTM (pseudoterminal master). A new PTS (pseudoterminal slave) is created in `/dev/pts/`³³, whose name can be discovered by providing the PTM file descriptor to `ptsname_r(3)`. Opening this PTS requires use of the `grantpt(3)` and `unlockpt(3)` calls. Once both sides have been opened, data freely flows between the PTM and PTS.

You generally shouldn’t need to mess with TTYs or PTYs yourself as an application developer, but now you know what all that junk clogging up your `/dev` is. Some day when you mess up your `initramfs`, and you’re getting an error message like “`/dev/ptmx` does not exist you have no chance to survive make your time” you’ll know you’re missing the master clone device, some small consolation as you hopelessly reboot³⁴.

6.1 Terminals and the UNIX process model

Understanding how these devices—physical or virtual—end up connected to actual processes requires understanding some basic details of the UNIX process model³⁵. Recall the UNIX principle that “everything is a file”. Generally—and this is true for all terminal devices—devices will be `open(2)`ed using their `/dev` nodes, and

³²You might hear about both System V and BSD pseudoterminals. SUS standardized around the System V form, known on Linux as “UNIX 98”. This term refers to 1997’s SUSv2 “`_(\)_/`”. Trash BSD `ptys` look like `/dev/ttyLN` and `/dev/ptyLN`.

³³`/dev/pts` is typically its own `devpts` filesystem on Linux.

³⁴Pro tip: should you ever need supply input to programs which refuse to read from pipes (e.g. `passwd`), you can pump the input through a named `pty`.

³⁵For a more complete description, consult Chapter 9 of [107].

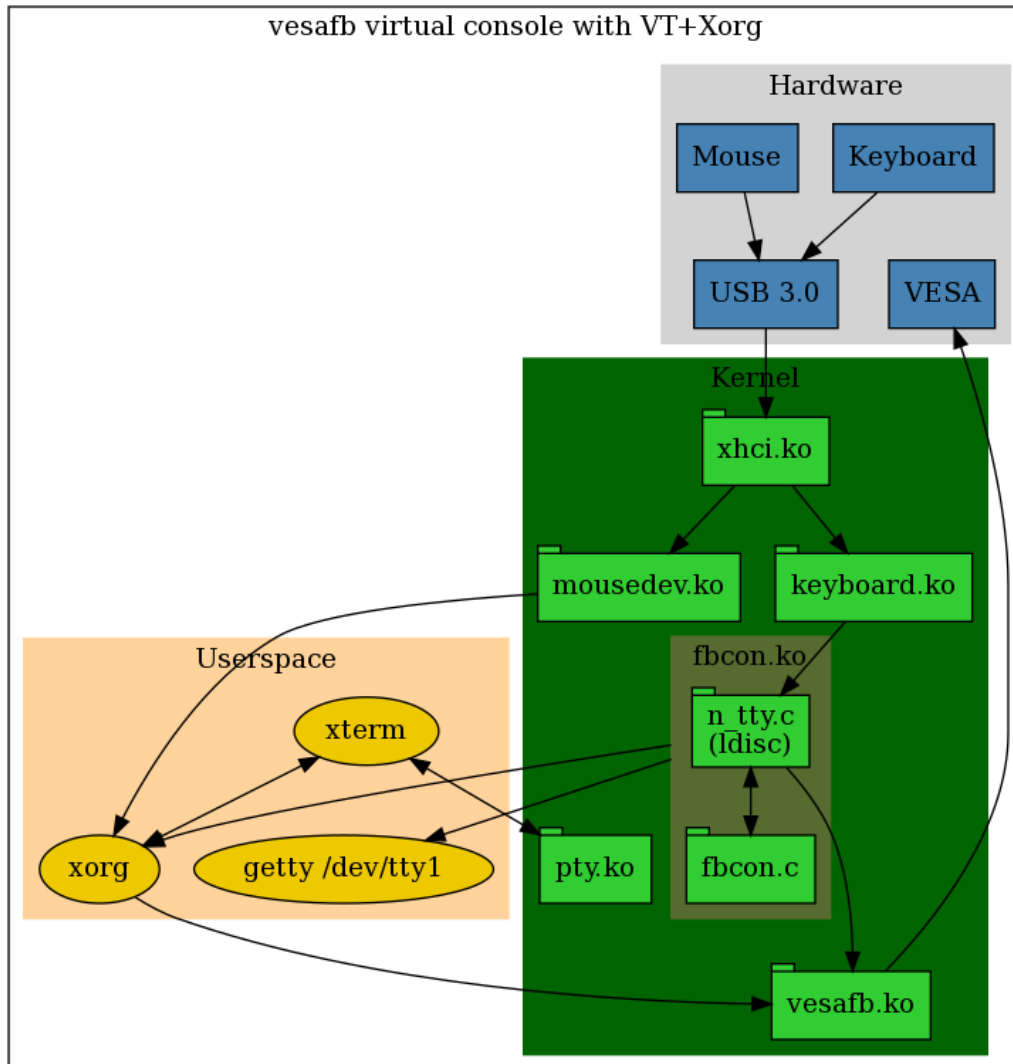


FIGURE 29: VESA + USB virtual console, from hardware to userspace. Adapted from [2].

then operated upon as a typical file descriptor. You're no doubt aware that UNIX programs traditionally start with at least three file descriptors open: 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`. Recall further that file descriptors are by default not closed across an `exec(2)` call that replaces one process's code with another object.

Each process has a process ID (PID) unique to its PID namespace (see `clone(2)` [17], particularly the `CLONE_NEWPID` flag). Each process is a member of a *process group*, and each process group has its own process group ID (PGID)³⁶. This PGID is equal to the *group leader's* PID, and each process group has one and only one leader. Signals can be addressed to a process group by supplying the negative PGID to e.g. `kill(2)`; a signal sent to a group will be delivered to all processes of that group. One of the most common scenarios in which multiple processes are kept in a single group is a shell pipeline (a `fork(2)`ing process will usually place independent children into a new process group). This is the essential facility underlying job control (placing a job into the background, etc.), explaining why backgrounding applies to an entire pipeline. Processes may be moved between process groups, but only within a *session*.

³⁶Both PIDs and PGIDs are represented using `pid_t`.

All process groups (and thus all processes) are members of some session. Each session has a session ID (SID)³⁷. This SID is the same as the PID of its *session leader*. A session might or might not have a controlling terminal (daemons disconnect themselves from their controlling terminal). Login sessions almost always do have a controlling terminal: the tty on which `login` was running (for a console session), or the pty opened at process creation (for e.g. SSH or `xterm`). The process groups of a session are divided into a single *foreground group* and zero or more *background groups*.

Figure 30 shows the terminal-connected processes on one of my Linux workstations, with their PIDs, PPIDs (parent process IDs), PGIDs and SIDs. `agetty` is listening on `tty1`, spawned by `init` (process 1). Xorg is sitting on `tty7`, having been spawned by a display manager (which is **not** itself connected to a terminal). The bash at PID 41599 is the child of a `screen` process. bash at PID 846497 is a child of `sshd`.

```
[killer mike] (0) $ ps --ppid 2 --deselect 0 pid,ppid,pgid,sid,TTY,comm | grep -v ?
  PID  PPID  PGID  SID TT  COMMAND
 41599 41598 41599 41599 pts/1  bash
 41654 41599 41654 41599 pts/1  rtorrent main
510181 1 510181 510181 tty1  agetty
510609 510187 510609 510609 tty7  Xorg
846497 846495 846497 846497 pts/0  bash
851850 846497 851850 846497 pts/0  ps
851851 846497 851850 846497 pts/0  grep
[killer mike] (0) $
```

FIGURE 30: Terminal-connected processes on a Linux machine.

Sessions affect how signals from the terminal are distributed. `SIGINT` and friends (signals generated due to the line discipline) are sent to the foreground process group. Only the foreground process can read from the terminal; if a background process attempts to do so, it is sent `SIGTTIN`, the default handler of which is to stop. A stopped background process will receive `SIGCONT` when brought into the foreground. Likewise, a background process will be sent `SIGTTOU` if it attempts to write to the terminal when the `TOSTOP` tty flag is valid, or attempts to reconfigure the terminal in any way^[86].

Finally, it's worth knowing how terminals get turned into login sessions. Under `systemd-logind`^[44], `agetty` processes are launched on demand (i.e. on first transition to the virtual console) on `ttyN` through the value of `NAutoVTs`. Should you be so unfortunate as to still be running SysVinit in 2020, ttys are configured in `/etc/inittab`³⁸. On FreeBSD, `/etc/ttys` plays a similar role^[113]. `getty`, `agetty` et al condition the line (if necessary), prompt for a login name, and then invoke `login` or a similar program. `login` prompts for a password (or `sshd` performs authentication by myriad means), and if the user is authenticated, `exec(2)`s a login shell as a new session. This login shell inherits the terminal file descriptors from parent processes, and passes them (by default) to its children.

Environment variables are also inherited across process boundaries, and by the time the login shell is run, the all-important `LANG` and `TERM` variables ought have been set. `LANG` defaults are based off the contents of `/etc/locale.conf`. `TERM` is generally set by the terminal emulator itself, or by `getty` on the console, and it is always forwarded by `ssh`³⁹. It's critical to have a correct `TERM` value, and misinformation on configuring it runs rampant. Besides the NCURSES FAQ^[37], I find the nosh project's `TERM` page^[13] to be an excellent resource, along of course with the `term(7)` man page^[110]. You will likely want to manually ensure that `COLORTERM` is properly defined for 24-bit TrueColor, assuming your terminal supports it.

6.2 Control sequences ANSI and otherwise

Quite early on, various hardware terminals began adding vendor-specific control sequences—special bit patterns which, when present in the output stream, would have effects other than simple emission of control

³⁷This session ID isn't really its own unique value—it's implied by the session leader process.

³⁸If you're one of those people who pine for SysVinit, your opinions are bad, and you should feel bad.

³⁹There is an exception; `ssh` does not forward `TERM` when the `-T` parameter explicitly inhibits pty allocation.

or graphic characters⁴⁰. These first took the form of “shifters”, which substituted regions of the code set (and form the basis of ISO 2022-style character set extension). With the advance of electric displays, it became desirable to apply systematic changes to the basic character set, and to more flexibly update the screen. The VT52 employed Bob Bemer’s ESC control character[9] to introduce certain control sequences, and by the time ANSI X3.41 rolled along in 1974 (to become ECMA-48 in 1976), it was reasonable to start all control sequences with ESC (having been made an official part of ASCII by ANSI X3.4-1968). The VT100 maintained compatibility with VT52-style escapes, but its 1970 User Guide already admonished programmers that “All new software should be designed around the VT100’s ANSI mode.”[30].

It is rare to encounter a terminal that implements no ANSI control sequences; I doubt any system exists which implements all of them⁴¹. Given the somewhat vague nature of ECMA-48 even in its most recent (1991) edition[5], it is unlikely that two terminals ever implement it the exact same way. Even assuming an ANSI terminal, it’s thus critical to use a terminal abstraction library (notwithstanding some people’s well-meaning claims[50]); see the next section for more information. Most of the useful ANSI sequences are prefixed by ESC[(0x27 0x5B), the “Control Sequence Initiator”. A selection of the ANSI control sequences is listed in Table 3 (but they should always be used through terminfo, as we’ll see in a moment).

Sequence	Name	Termcap	Description
c	RIS	rs1	Reset to original state
[nA	CUU	cuu	Move up <i>n</i> rows
[nB	CUD	cud	Move down <i>n</i> rows
[nC	CUF	cuf	Move right <i>n</i> columns
[nD	CUB	cub	Move left <i>n</i> columns
[val...valm	SGR	sgr	Set graphic rendition

TABLE 3: A few ANSI (ECMA-48) escape codes. All sequences follow an ESC.

Note that some escapes commonly misattributed to ANSI come instead from vendor extensions. The Restore Saved Cursor (RCO) and Save Cursor Position (SCO) escapes, for instance, show up all over the Internet as ANSI escape sequences, but no trace of them can be found in ECMA-48:1991.

In a relic from teletypes, ISO 646, ISO 2022, and ECMA-35 described use of non-destructive backspace to produce composed characters from spacing ones. This had been eliminated by ISO 4873, ECMA-43, and ISO 8859.

6.3 Terminfo

While developing the editor *ex*, a precursor to the great *vi*, Bill Joy implemented for BSD a database and set of functions abstracting away the differences between terminal emulators (inspired by similar capabilities of MIT’s Incompatible Timesharing System[32]). The terminal abstraction layer was ready-made for release as the original termcap in 1968 (Ken Arnold would a few years later genericize the cursor optimization routines as the original BSD Curses[4]). The termcap database (*/etc/termcap*) is indexed by terminal type and two-letter property name, and maps them to boolean, numeric, and string capabilities.

Termcap has a number of limitations, and its use in new programs is strongly discouraged. In its place is recommended terminfo, a more powerful scheme originally introduced as part of Mary Ann Horton’s⁴² AT&T curses[54]. Notcurses makes extensive use of terminfo, drawing from it all escapes save RGB. The terminfo database and API are specified in SUS, much like Curses (and indeed the libtinfo installed on most Linux machines is shipped as part of NCURSES). The terminfo database can be dumped with *infocmp(1)*, and specific properties can be queried with *tput(1)*. *tset(1)*, *clear(1)*, and *reset(1)* are usually wrappers around terminfo.

⁴⁰How early? The Baudot code of 1870 already had a “Letter shift”, and the Teletypesetter news wire code of 1928 introduced “Shift in” and “Shift out“. Pretty goddamned early, then.

⁴¹There’s some crazy stuff in ECMA-48. “Media copy”? “Disable manual input”? Many things are woefully underspecified.

⁴²Formerly known as Mark Horton.

The `stty` tool accidentally duplicates some functionality of `terminfo`, but it does not itself make use of `terminfo`. It is more accurate to describe `stty` as a command-line interface to the `termios` routines `tcgetattr()` and `tcsetattr()`. Dump all terminal attributes to the screen with `stty -a`⁴³.

bash/X c1/seat0 tty1 sid 14167	login	14167	1	tty1
	bash	15189	14167	tty1
	xinit	28146	28124	tty1
	Xorg	28147	28146	tty1
login tty3 sid 450305	login	450305	1	tty3
xterm sid 1332684 sid 2888258	compiz	28209	28147	?
	xterm	1332685	28209	?
	bash	2888258	1332685	pts/1
Local				
bash/ssh c6 sid 3175763 sid 3175778	sshd	14045	1	null
	sshd	3175763	14045	null
	sshd	3175775	3175763	null
	bash	3175778	3175775	pts/2
screen sid 1877617	screen	1877617	1	null
	bash	1877618	1877617	pts/7
Detached/Daemons				
Sessions, seats, processes, TTYs, and FDs Debian Unstable (no display manager)				

FIGURE 31: Processes, sessions, PPIDs, and TTYs. Each grey box is a lineage of processes. Session IDs are listed. Columns are process names, PIDs, PPIDs, and open TTY.

⁴³Pro tip: `stty` generally works only on standard input...but redirecting standard input from a `/dev` node works just fine (subject to permissions, of course).

7 Character encodings and glyphs

Thou whoreson zed, thou unnecessary letter!

Wiliam Shakespeare, *King Lear*

We'll be well-served by becoming familiar with our core building blocks—character sets and their rasterized forms. Even if we were to shrink the character cell to a single pixel, we'd still need write one or another character into it, and emit a control code to stylize it⁴⁴.

In the beginning, there are somewhat Platonic *characters*, one of the most imprecise terms in all of computer science (even when we leave aside the data type `char`). In and of itself, a character is nothing more than an identifier: LATIN CAPITAL LETTER I⁴⁵. There are many things a character is not:

- A character is not necessarily unique within a character set (see diacritic vs. precomposed forms).
- Different characters needn't be distinct glyphs[119].
- A character needn't limit itself to a single column, even when using a fixed-width font.
- A character does not necessarily have a visual representation.
- A character does not necessarily have a single meaning among different languages.
- In a given encoding, all characters are not necessarily the same size.
- A character cannot necessarily capture the state of the keyboard at some time.
- A character cannot necessarily describe a cell of a display at some time.

Collecting characters and assigning them distinct (but not necessarily contiguous) numbers creates a *code set*, and the smallest contiguous range of numbers covering this code set is a *code space*. The Unicode 13 code space is comprised of 17 contiguous planes of 65,536 code points each, for a size of $17 * 2^{16}$. It defines 143,859 characters, smaller than its code space by an order of magnitude. A map from a code set to a set of bit sequences is a *character encoding*. There are numerous standard encodings of the Universal Character Set, each of which represents the entire set in different ways (the best one is UTF-8, as we will see). Character sets are registered with IANA per RFC 2978[45], and character encodings are registered with the ISO Register of Coded Characters per ISO/IEC 2375[63].

Character graphics are formed from *glyphs*, the visual renderings of a character encoding, supplied by a font. These glyphs might fuse into what Unicode Standard Annex #29[25] refers to as a *user-perceived character* or *grapheme cluster*. A formal algorithm for dividing a stream of glyphs into grapheme clusters (“horizontally segmentable units of text[49]”) achieves *segmentation*. The current Unicode segmentation algorithm yields *extended grapheme clusters*.

These extended grapheme clusters (EGCs) can be thought of as the atoms of character graphics. It is not generally possible to write them partially, nor to partially overlay one atop another. Backspacing ought usually remove entire EGCs at a time. The cursor ought cross an EGC in a single movement. In Notcurses, each cell of a plane can hold a single EGC, which must be written as a single unit. The actual number of columns occupied by an EGC is a property of the font and layout engine being used.

In the modern era, Unicode is used just about everywhere, and programs must be prepared to handle it. On the plus side, Unicode presents a single, coherent means of representing all the world's languages, eliminating the old necessity of switching among encodings at runtime. UNIX has its origins—and most of its modern UI—in a much smaller character set: ASCII.

⁴⁴I've seen this idea bandied about as if it's a serious suggestion. Besides the fact that it won't work in a console, escapes are *far* less efficient than canvases or OpenGL display lists; if your plan involves using ANSI escape sequences to drive a 1920x1080 display via roughly two million 1x1 character cells, you're not going to space today, or maybe ever[84].

⁴⁵Isn't that a recursive definition? You betcha.

7.1 Everyone loves ASCII

The first digital codes were the English telegraph system of Charles Wheatstone and William Cooke, and American Samuel Morse’s code, with which he in 1837 famously signaled “What hath God wrought!” The Morse system was simpler, running over a single line, and quickly won out. The teleprinter of Jean-Maurice-Émile Baudot (from whom we derive the modern unit *baud*) followed in 1874[42], with its pentabit, fixed-length Baudot code. Baudot code required 7.42 bittimes per character; at a typical 0.022s bittime, it could transmit just over 6 characters per second[81].

For our purposes, the story can reasonably start with a 7-bit encoding of 128 characters: ANSI X3.4-1986, perhaps better known as ASCII (Table 4). The first ASCII we would recognize as such⁴⁶ was the unpublished ASCII-1965. The ASA (ANSI before it was ANSI) ASCII 1963 didn’t even have minuscules⁴⁷. ASCII-1968 was released as USAS X3.4-1968 to wild acclaim. The ISO/IEC 646 standard[66] “internationalized” ASCII-1968 by opening up 12 graphic characters to regional specifications (the US ASCII defined the “International Reference Variant”)[87].

The first 32 values are non-printable control codes, first given their current definitions in ASCII 1968. These 32 codes (known as the “C0” coded control set since ISO/IEC 646) lived on through ISO/IEC 2022 (ECMA-35), ISO/IEC 6429 (ECMA-48), and ISO/IEC 8859 (ECMA-94), and are reproduced in today’s ISO/IEC 10646, aka the Universal Character Set. They’re common across just about every character set of which I’m aware (EBCDIC went almost entirely its own way, because EBCDIC), despite being largely archaic and altogether mystifying. Most of the associated semantics have been obsoleted, and in some cases the encoded characters are now used for different purposes than originally planned (see Table 5). These characters can be entered by pressing Ctrl along with another key from ASCII; these combinations are defined using “caret notation”.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

TABLE 4: ANSI X3.4-1986 (ISO/IEC 646-IRV, IA5, T.50 IRA, RFC 20)—American Standard Code for Information Interchange. Shaded characters can be replaced in regional variants.

⁴⁶To paraphrase G. H. Hardy, “ASCII-1963 was clever schoolboys; ’68 a Fellow from another College[51].”

⁴⁷Uppercase and lowercase are loanwords from printing, where the two sets were stored in different cases of the typesetting table. “Majuscules” and “minuscules” will mark your good breeding.

In the early 1960s, an IBM employee on a dare ate a hallucinogenic tapeworm that pissed nitric acid. Stuffing his steaming, rapidly decoiling bowels up into a brown paper bag, he was off to the hospital. Alas! On those very hospital steps he was struck by lightning, and trepanned with great drills, and set upon by savage hogs, and finally exploded. The tapeworm emerged, clad in shimmering mandorla, with a name pronounceable by no human tongue. “Adding hallucinogenic tapeworms to a late project only makes it later,” cautioned Fred Brooks[14], but Gene Amdahl knew no peace. “This Tapeworm Queen reminds me of halcyon Wisconsin days!” he cried, and on a prototype Model 30 appeared her Name. Struck by the power of that Name, all began to bow, and to weep. She vomited forth the Tapeworm Gospel. Looking upon this filth, they thought it Good. So now you know what EBCDIC is. There ended up being no fewer than 57 variants[99] of EBCDIC, worthy of Heinz. We will speak of it no further. If you find yourself needing more information about EBCDIC, think long and hard about your life decisions.

Recall the `struct termios` from Chapter 6. The `c_cc` array of that structure defines “special characters” for the terminal. Aside from CR and LF, the various caret controls can there be recovered and redefined. Should you really want to send `SIGINT` via ‘a’ rather than Ctrl+C, that’s where you can do it.

Character	Caret	C	c_cc	Semantics
0x00 (NUL)	^@			
0x03 (ETX)	^C		VINTR	Deliver <code>SIGINT</code>
0x04 (EOT)	^D		VEOF	Indicate end-of-file
0x07 (BEL)	^G	\a		Alert (bell)
0x08 (BS)	^H	\b		Backspace
0x09 (HT)	^I	\t		Proceed to next tab stop
0x0a (LF)	^J	\n		Move down one line
0x0b (VT)	^K	\v		Move down to next vertical tab
0x0c (FF)	^L	\f		Move to start of line
0x0d (CR)	^M			Carriage return
0x0e (SO)	^N			(ISO 2022 mode) SO/LS1
0x0f (SI)	^O		VDISCARD	(ISO 2022 mode) SI/LS0
0x11 (DC1)	^Q		VSTART	Software flow control (resume output)
0x12 (DC2)	^R		VREPRINT	Reprint input prompt
0x13 (DC3)	^S		VSTOP	Software flow control (pause output)
0x15 (NAK)	^U		VKILL	Erase line
0x16 (SYN)	^V		VLNEXT	Literal next—inhibit translation
0x17 (ETB)	^W		VWERASE	Erase word
0x1a (SUB)	^Z		VSUSP	Deliver <code>SIGTSTP</code>
0x1b (ESC)	^[Initiate escape sequence
0x1c (FS)	^\		VQUIT	Deliver <code>SIGQUIT</code>
0x1d (GS)	^]			
0x1e (RS)	^^			
0x1f (US)	^_			

TABLE 5: Usual UNIX semantics of C0. The exact mappings can be inspected with `stty -a`.

From Table 5, it should be obvious that Ctrl essentially clears the top two bits of a 7-bit ASCII input. It might be less obvious that all minuscules are equal to the sum of 0x20 and their corresponding majuscule, allowing lowercase and uppercase to be switched by toggling the fifth (32) bit. Likewise, 0xDF provides a fast case-insensitive eight-bit comparison mask (it is necessary to mask out 0x2). Numeric values for a digit can be acquired by subtracting 0x30 ('0'). Printable characters are all those with a 1 in any of the higher five bits, save 0x7F⁴⁸. Numeric values for a letter can be acquired by subtracting 0x61 ('a') from its minuscule form. The digits are contiguous, as are each case of letters⁴⁹.

7.2 Octa- and hexabit character sets messily diverge

US-ASCII and the regional dialects of ISO/IEC 646 were more or less sufficient for working with English⁵⁰, but even the accented Romance scripts needed more room to be expressed. Other segmentally linear, monophonemic writing systems (e.g. Cyrillic, Hangul, or Greek) would need replace the graphic characters wholesale. The idea of enumerating syllabary-based languages in seven or even eight bits is laughable⁵¹.

By the late 1970s, the eight-bit “octet” byte could be considered dominant⁵². The very first ANSI/ISO C

⁴⁸Why 0x7f? Like so many things, this comes down to punch cards. If you’ve punched an error on a group of 7 holes, and want to correct it, what are your choices? The only general solution is to interpret all 7 holes as a `strikeout`[108].

⁴⁹This is not true for EBCDIC, where the letter sequences are based on the “zones” of punch cards[47].

⁵⁰As far as this author knows, ASCII can faithfully represent only (modern) English, Latin, and Swahili.

⁵¹An octet character set for “rudimentary form” Japanese, containing only the *katakana*, was introduced as JIS X 0201-1976.

⁵²Wikipedia would have you believe (as of 2020-03, anyway) that the System/360 introduced the eight-bit byte. The Sys-

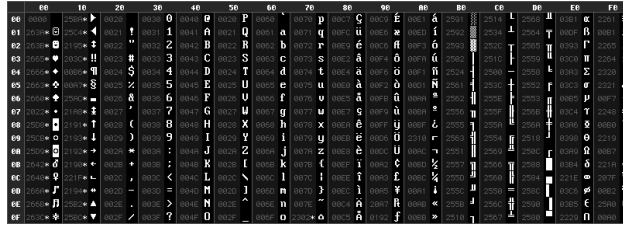


FIGURE 32: Code Page 00437, the IBM PC font immortalized by VGA (Source: *VileR* under CCA3.0).

standard (1989) established `CHAR_BIT` to have a minimum value of 8⁵³. With the high bit of an octet byte going largely unused as a parity bit, the seven-bit character sets were rapidly expanded into a wide variety of eight-bit character sets (sometimes mistakenly referred to as “extended” or “high” ASCII—the closest thing to “extended ASCII” might be the withdrawn ANSEL ISO-IR 231 “ANSI extended Latin”^[58]).

Among the better-known ASCII-derived eight-bit “code pages” was IBM’s 00437, released with the 5150 PC. It spread far and wide by virtue of being burned into the ROM of IBM’s MDA (9x14), CGA (8x8), EGA (8x14), and VGA (9x16) controllers. CP437 used most of the extended space for semigraphics, mathematical symbols, and some characters useful in the world as perceived by America circa 1981⁵⁴—CP437 has you covered regarding the Dutch florin and the Spanish peseta⁵⁵. It also introduced the non-breaking space (at code 0xff). Interestingly, a melange of semigraphics were assigned to the C0 control character code points, though these retained their previous meanings (sometimes). All CP00437 characters have similar glyphs in ISO 10646, though not necessarily at the same locations⁵⁶. Derivatives of CP437 usually jettisoned the mixed box drawing characters and some of the math symbols to further expand alphabet coverage.

There are well over a thousand known code pages corresponding to regional, corporate, and even private character sets, and I shall not even begin to cover them all. There are code pages by Microsoft popularly known as the “ANSI code pages” (despite not being based on any ANSI standard), and there are code pages by IBM for Windows emulation which differ from the Windows code pages. HTML5 requires that “text/” media types (that would previously have been interpreted as ISO-8859-1) be interpreted as Windows-1252; Windows refers to ISO-8859-1 as Windows-28591. Be glad that you are not working in the age of 8-bit character sets. This discordant cacophony gave rise to *mojibake* (文字化け [*modzibake*]), the garbled result of mismatched character encoding and decoding⁵⁷. ISO 4873^[65] formalized a split between control character sets (prefixed with ‘C’) and graphic character sets (prefixed with ‘G’); ISO 2022 formalized a baroque mechanism for shift-

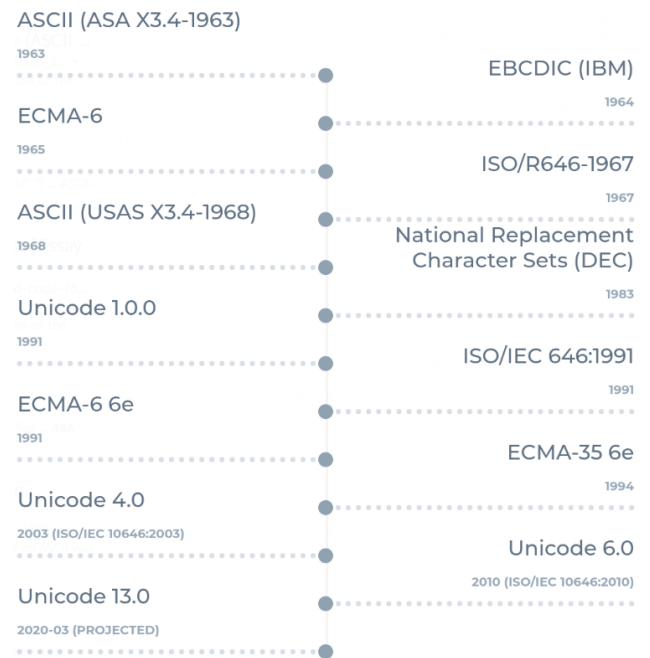


FIGURE 33: Timeline of selected sets.

tem/360 introduced a great many things, but according to my own research, the IBM Stretch 7030 of 1961 used an 8-bit data path three years prior^[106]. The last major sub-octet processor was probably the Intel 4004, and while the PDP-8 was a 12-bit machine, it often operated on 6-bit characters.

⁵³It can have a larger value than 8, of course, and indeed does on many DSPs^[71]. POSIX does mandate that `CHAR_BIT==8`.

⁵⁴The only suggestion that Asia existed as of CP437 is the ¥.

⁵⁵Which said peseta didn’t even have a symbol, just ₧.

⁵⁶A few of the CP437 characters have multiple possible UCS equivalents; selecting the correct one depends on context.

⁵⁷Japanese, from 文字 (moji (character)) 化ける (bakeru (take a different form)). Russians know it as *кракозябры* (*krakozyabry* [*kɾɛkɐˈzʲæbrʲi*]) or sometimes БНОПНЯ (*bnopnja* *bnɐpˈnʲa*), while Bulgarians speak of *маймуница* (*majmunica* (monkey alphabet)). Serbs cut to a characteristically Balkan chase with *ћубре* (*đubre* (trash)). I ♠ Unicode.

ing between “left” (for handling 7-bit graphic characters) and “right” (for handling 8-bit graphic characters) active graphic character sets from a selection of G0–G3. Really, be quite glad that you are not working in the age of 8-bit character sets.

From a historical perspective, the most important eight-bit character set is likely ISO-8859-1[67], based on the Multinational Character Set of DEC’s VT220, ANSI X3.4-1986, and the C1 control code set defined in ISO 6429. Despite being a terrible attempt at encompassing the French alphabet⁵⁸, ISO-8859-1 was directly mapped to the Basic Latin and Latin-1 Supplement blocks of UCS, together having range 0–0xff. Thus the first 128 codepoints of both ISO-8859-1 (all the ISO 8859 sets, actually) and UCS are inherited from ANSI X3.4-1986 + ISO 646’s C0 + ISO 2022’s universal declaration of SP (space, 0x20) and DEL (delete, 0x7f), and the second 128 codepoints of UCS are inherited from ISO-8859-1 + ISO 6429’s C1. Only the *codepoints* are equal, however—these codepoints are not typically encoded the same way⁵⁹.

There were also some true 16-bit and multibyte character sets, mainly in East Asia. You might hear of Shift JIS, EUC-JP, Big5, and GB18030. How unfortunate.

7.3 Consilience: the Universal Character Set

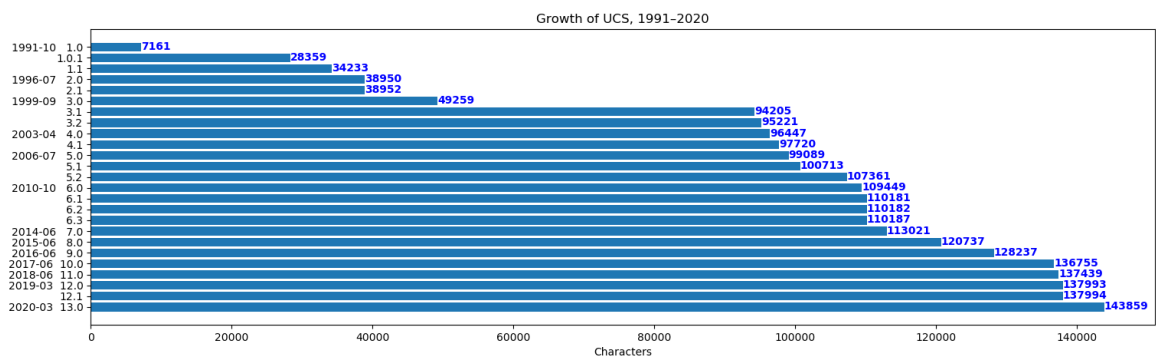


FIGURE 34: 2020’s Unicode 13.0 ships 143,859 character definitions.

Principle	Statement
Universality	The Unicode Standard provides a single, universal repertoire.
Efficiency	Unicode text is simple to parse and process.
Characters, not glyphs	The Unicode Standard encodes characters, not glyphs.
Semantics	Characters have well-defined semantics.
Plain text	Unicode characters represent plain text.
Logical order	The default for memory representation is logical order.
Unification	The Unicode Standard unifies duplicate characters within scripts across languages.
Dynamic composition	Accented forms can be dynamically composed.
Stability	Characters, once assigned, cannot be reassigned and key properties are immutable.
Convertibility	Accurate convertibility is guaranteed between the Unicode Standard and other widely accepted standards.

TABLE 6: The 10 Design Principles of Unicode (Unicode Core Specification §2.2[23]).

The Universal Character Set (ISO 10646⁶⁰, the character set underlying Unicode) was introduced to resolve

⁵⁸French discontent eventually resulted in the “corrected” ISO-8859-15, published in 1999, by which time Unicode was eight years old and no one really gave a *merde*[68].

⁵⁹A notable and critical exception is UTF-8, which encodes the first 128 codepoints the same way as ISO 646. Read on...

⁶⁰Note that this is 10,000 more than ISO 646. Cute?

these problems, as a “unique, universal, and uniform character encoding”[20]. The Unicode Consortium traces its origin to Joe Becker’s paper, “Unicode 88”[8]. Often in these early documents, UCS is described as a “16-bit character set”. This is perhaps partially responsible for the widespread misconception that UTF-16[53] can encode all UCS code points in a single 16-bit unit. 16 bits are sufficient to encode any given UCS *plane* of up to 65,536 code points, and indeed a great many languages are contained within the first UCS plane, the Basic Multilingual Plane (0x0–0xffff). There are a total of 17 planes available in UCS⁶¹, however, and UTF-16 requires two code units to encode points from these other 16 planes (0x10000–0x10ffff).

Plane	ID	Purpose
Basic Multilingual	0	Common-use characters for modern and historical scripts.
Supplementary Multilingual	1	Spillover from BMP.
Supplementary Ideographic	2	CJK spillover from BMP.
Supplementary special-purpose	14	Format control spillover.
Supplementary Private Use A	15	Local semantics.
Supplementary Private Use B	16	Local semantics.

TABLE 7: The six named UCS planes (Unicode Core Specification §2.8[23]).

Below the plane comes the *block*, a range of between 16 and 65,536 related code points, always starting on a 16-codepoint boundary. A block is merely a convenience for reference. Each codepoint belongs to one and only one block. As mentioned earlier, the first 256 code points of UCS (making up the first two blocks) are taken from ISO-8859-1, plus DEL and the control character classes C0 and C1.

	2010 1 Jan	2011 1 Jan	2012 1 Jan	2013 1 Jan	2014 1 Jan	2015 1 Jan	2016 1 Jan	2017 1 Jan	2018 1 Jan	2019 1 Jan	2020 1 Jan	2020 17 Mar
UTF-8	50.6%	59.8%	68.0%	74.7%	78.7%	82.3%	86.0%	88.2%	90.5%	92.8%	94.6%	95.1%
ISO-8859-1	28.6%	22.0%	17.2%	13.5%	10.8%	9.3%	6.9%	5.5%	4.3%	3.6%	2.6%	2.3%
Windows-1251	4.3%	3.7%	3.3%	2.8%	2.7%	2.2%	1.9%	1.7%	1.5%	1.1%	1.0%	1.0%
Windows-1252	3.2%	2.3%	1.7%	1.3%	1.3%	1.1%	1.0%	0.9%	0.7%	0.7%	0.5%	0.4%
Shift JIS	3.1%	2.2%	1.7%	1.4%	1.4%	1.3%	1.1%	1.0%	0.8%	0.4%	0.3%	0.2%
GB2312	3.5%	4.4%	3.6%	2.5%	2.0%	1.4%	0.9%	0.8%	0.6%	0.4%	0.2%	0.2%
EUC-KR	0.3%	0.2%	0.2%	0.2%	0.2%	0.4%	0.4%	0.4%	0.3%	0.3%	0.2%	0.1%
ISO-8859-9	0.7%	0.5%	0.3%	0.3%	0.2%	0.2%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
EUC-JP	0.7%	0.5%	0.4%	0.4%	0.4%	0.3%	0.3%	0.3%	0.2%	0.1%	0.1%	0.1%
Windows-1254	0.4%	0.3%	0.2%	0.2%	0.2%	0.1%	0.1%	0.1%	0.1%	<0.1%	0.1%	0.1%
GBK	0.7%	0.9%	0.9%	0.8%	0.6%	0.4%	0.3%	0.3%	0.2%	0.1%	0.1%	0.1%
Big5	0.4%	0.3%	0.2%	0.2%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
ISO-8859-2	0.9%	0.7%	0.6%	0.5%	0.4%	0.3%	0.3%	0.2%	0.2%	0.1%	0.1%	0.1%
Windows-1250	0.4%	0.3%	0.3%	0.2%	0.2%	0.2%	0.2%	0.2%	0.1%	0.1%	0.1%	0.1%
Windows-1256	1.2%	1.2%	0.8%	0.5%	0.3%	0.2%	0.2%	0.1%	0.1%	0.1%	<0.1%	0.1%
Windows-874	0.2%	0.2%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%
ISO-8859-15	0.5%	0.4%	0.4%	0.4%	0.3%	0.2%	0.2%	0.1%	0.1%	0.1%	0.1%	<0.1%
US-ASCII	0.2%	0.1%	0.1%	0.1%	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
Windows-1255	0.1%	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
TIS-620	0.1%	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
ISO-8859-7	0.1%	0.1%	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
Windows-1253	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%

FIGURE 35: Charsets served over the last decade on the Web (*source: W3Techs*).

7.4 Fixed-width fonts ain't so fixed

Notcurses assumes that all glyphs occupy widths which are an integral multiple of the smallest possible glyph’s cell width (aka a “fixed-width font”). Unicode introduces characters which generally occupy two such cells, known as wide characters (though in the end, width of a glyph is a property of the font). It is not possible to print half of such a glyph, nor is it generally possible to print a wide glyph on the last column of a terminal.

Notcurses does not consider it an error to place a wide character on the last column of a line. It will obliterate any content which was in that cell, but will not itself be rendered. The default content will not be reproduced in such a cell, either. When any character is placed atop a wide character’s left or right half, the wide character is obliterated in its entirety. When a wide character is placed, any character under its

⁶¹Once upon a time, these other sixteen planes were known as the Astral Planes[85].

left or right side is annihilated, including wide characters. It is thus possible for two wide characters to sit at columns 0 and 2, and for both to be obliterated by a single wide character placed at column 1.

Likewise, when rendering, a plane which would partially obstruct a wide glyph prevents it from being rendered entirely. A pathological case would be that of a terminal n columns in width, containing $n - 1$ planes, each 2 columns wide. The planes are placed at offsets $[0 \dots n - 2]$. Each plane is above the plane to its left, and each plane contains a single wide character. Were this to be rendered, only the rightmost glyph would be visible!

Finally, fonts and font engines which yield glyphs wider (or narrower) than `wcwidth()` would lead Notcurses to believe can cause problems. It is best to avoid EGCs known to be very wide, or to avoid fonts which generate very wide glyphs. Some examples are shown in Figure 36.

7.5 Emoji

According to Unicode Standard Annex #51[26], emoji are “pictorial symbols that are typically presented in a colorful cartoon form and used inline in text. They represent things such as faces, weather, vehicles and buildings, food and drink, animals and plants, or icons that represent emotions, feelings, or activities.” Don’t blame me, mang; I didn’t do it. Emoji (Japanese 文字 え も じ ([`emodzi`]), 絵 (picture) and our old friend 文字 (moji (character)) emerged from Japan in the late 90s, and by June 2019 they were in the Oxford Fucking Dictionary of the English Language[36]—sorry, I get kinda touchy about the OED. Emoji represent, by far, the most kibbitzed-upon, politicized, arbitrary element of Unicode; for us, they are also the most dangerous. Presentation of emoji is rivaled only by bidirectional text for variation across platforms.

Some Unicode “emoji” support only a textual presentation (`text-only`), while some support both (with different defaults (`text-default/emoji-default`) for different characters). Selecting the presentation can sometimes be done by following the base with `U+FE0E VARIATION SELECTOR-15` (for text) or `U+FE0F VARIATION SELECTOR-16` (for emoji). The full data on such things is in the Unicode-adjacent “Data files for emoji characters”, but how closely your local implementation will conform this week is generally unknowable.

When using the textual presentation, emoji can be arbitrarily colored according to the standard Notcurses mechanisms. Emoji presentation will typically make use of its own colors (the background color will usually be honored, in my experience). The Fitzpatrick dermatological scale has been adopted for modifying the skin tone of base glyphs via Zero-Width-Joiner (ZWJ) sequences. Characters which can be so modified are designated with the `Emoji_Modifier_Base` property, and can be affected by characters with the `Emoji_Modifier` property. Both (and the ZWJ character) are part of the same EGC. “Standard” ZWJ sequences are described as “Recommended for General Interchange” (RGI) in the Unicode data files.

Official Unicode support for the concept of emoji was added in Unicode 6.0, but some earlier characters have been retroactively recognized as emoji, reaching all the way back to the 23 Zapf Dingbats of Unicode 1.0. Unicode 5.2 added 88 symbols for conformance with the Japanese Association of Radio Industries and Businesses (ARIB). The remaining 608 characters necessary to cover various Japanese mobile carriers were added to 6.0 (and indeed their definitions are given in terms of the Shift-JIS encoding[21]). Unicode 6.1 added the concept of emoji presentation.

Google, Apple, and other Californian megacorporations think 🍷 is the best representation for 🔫 `U+1F52B PISTOL` (I personally like 🏹). That’ll be important for things like 🏹🔫👤, an example used in #51 to illustrate the importance of maintaining pictograph direction. It’s a madhouse out there.

You might have seen what appeared to be emoji flags of the world. The Unicode Consortium, in their wisdom, wanted nothing to do with such geopolitical questions. Instead, the 26 “Regional Indicator Symbols”—one for each letter of the ISO basic Latin alphabet[66]—can be used to encode ISO 3166-1[64] two-letter country codes[121]. Some fonts contain flag glyphs, and some layout engines will map pairs of these symbols to flags. The UCS *does not* contain national “flag emoji” (it does have some other flags, though—see Table 8⁶²).

There are also official and non-official EGCs that commonly render flags, including the Jolly Roger (`U+1F3F4`, `U+2620`, 🏴), Gilbert Baker’s rainbow flag (`U+1F3F3`, `U+1F308`, 🏳️) and Monica Helms’s transgender flag

⁶²Using a green block, it is possible to generate the flag of the Great Socialist People’s Libyan Arab Jamahiriya through 2011.

Character	Version	Comments
☒ U+1F38C CROSSED FLAGS	Unicode 6.0	why Japanese crossed flags?
▣ U+1F3C1 CHEQUERED FLAG	Unicode 6.0	“chequered” whatever
▧ U+1F3F3 WAVING WHITE FLAG	Unicode 7.0	surrender!
▩ U+1F3F4 WAVING BLACK FLAG	Unicode 7.0	the Nick Black flag amirite
🚩 U+1F6A9 TRIANGULAR FLAG ON POST	Unicode 6.0	suspiciously golf-related

TABLE 8: Flags in Unicode.

The downside of using such complexly combined EGCs is of course that what’s intended to be a single EGC can be broken into multiple EGCs by the font rendering system. Suddenly, instead of ☒ you’ve got an inscrutable ☒. To add insult to injury, this will almost certainly screw up geometry through the end of the line, since Notcurses will have an incorrect concept of the cursor’s horizontal position. This assumes that the font in use has even the basic component glyphs—in the very limited Linux console font, it’s unlikely that anything good can come of using composed emoji. When used properly under aligned stars, however, emoji can make your TUI pop like nothing else.

Some applications map emoticons to emoji. This is not performed at the terminal level for any terminal of which I am aware, so if you want this feature, you’ll need effect it yourself. Notcurses provides no support for this, because I think it’s dumb. There is no concept of emoticons in Unicode.

7.6 Stupid Unicode tricks

Perhaps the single most useful Unicode character is U+2580 UPPER HALF BLOCK (▀), or alternatively its inverse U+2584 LOWER HALF BLOCK (▁). By using these together with both a foreground and background, it is possible to treat a region of cells as a “framebuffer” having vertical resolution twice the number of occupied lines, and perhaps more importantly having a cell aspect ratio of 1:1. With this technique, it’s trivial to blit RGBA as provided by e.g. image decoders and get good results (this is exactly what’s done in Notcurses’s media layer, see Chapter 11). A smart implementation will, when possible, replace a HALF BLOCK plus two RGB specifications with U+2588 FULL BLOCK and a single foreground RGB. A smarter implementation still will, when possible, replace a FULL BLOCK with a space and a single background RGB⁶³, though this optimization saves only 2 bytes compared to the dozen or so saved by eliding an RGB escape. Notcurses effects both optimizations.

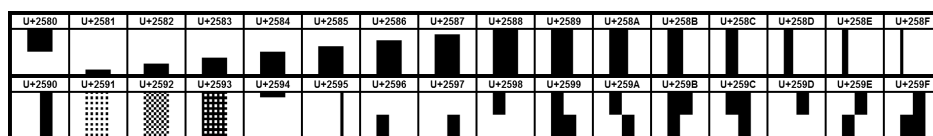


FIGURE 37: Unicode 13.0 Block Elements (source: Antonsusi under CCA3.0).

Unfortunately, it’s difficult to take this technique any further. There are U+258C LEFT HALF BLOCK (◀) and U+2590 RIGHT HALF BLOCK (▶): they exacerbate the character cell aspect ratio problem, but can still be useful for rendering certain surfaces in tall, thin geometries. A single cell can be divided up to 8 ways using the eighths blocks, but only being able to supply two colors (not to mention the resulting aspect ratio) means this can’t be easily used for 8x resolution. There are three incompletely-filled shades of U+2588 FULL BLOCK, which could conceivably be used to increase the color range by a factor of 4, but the 24-bit RGB space is already pretty large—resolution is a much more precious resource.

Those readers who remember Code Page 437 might recall 0xFD, a “middle half block”. Given a constant background, this can be used with the other two half blocks to increase “vertical resolution” by a factor of 3 (imagine a tank drawn with upper, middle, and lower half blocks moving on a monochromatic background).

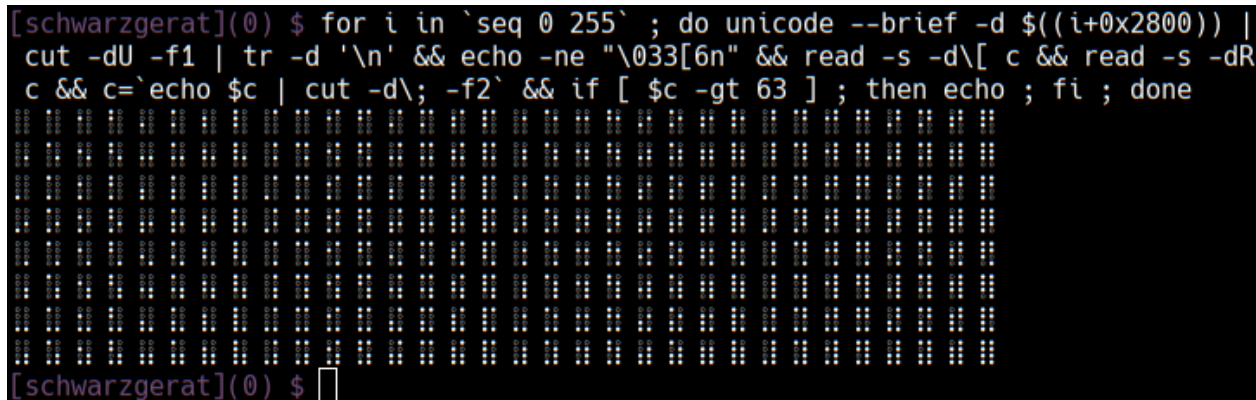
⁶³This optimization is not possible when, for instance, the alphas are anything other than CELL_ALPHA_OPAQUE.

Well, that character was actually “Solid Square/Histogram/Square Bullet” in the CP437 definitions[31], and lives on as U+25A0 BLACK SQUARE (■). Despite its name, it will take on the foreground color.

It might be useful to use the quadrant characters to effect a 2x2 increase in resolution. The quadrants cannot yield a lossless reproduction for any 2x2 block that uses more than two colors, but it might be useful to interpolate given the small total number of possible colors. Some graduate student ought look into this.

It is possible to use the Braille Patterns at U+2800 to get an 8x increase in resolution, at the cost of having only two colors available per 2x4 “8-pixel” cell (and looking like blotter paper, depending on the font). This can be an excellent solution for e.g. graphing, especially vertical histograms. By inspecting Figure 38, it ought be clear that the 8 areas map to distinct bits: counterclockwise from the upper left, the bits are 1, 2, 4, 64, 128, 32, 16, 8. This odd ordering is due to true braille only being a 6-bit (64 character) character set; this way, real braille maps to the first 64 codepoints.

```
[schwarzgerat](0) $ for i in `seq 0 255` ; do unicode --brief -d $((i+0x2800)) |
cut -dU -f1 | tr -d '\n' && echo -ne "\033[6n" && read -s -d\[ c && read -s -dR
c && c=`echo $c | cut -d\; -f2` && if [ $c -gt 63 ] ; then echo ; fi ; done
```



```
[schwarzgerat](0) $
```

FIGURE 38: Unicode Braille characters.

Certain sets of Unicode characters form cyclic groups. These can be useful for simple, single-cell animations, e.g. work-indicating “spinners”. I’ve isolated some of these groups in Table 9. Smaller groups can usually be isolated from larger groups: as an example, the eight-way HALF-SQUARE cycle yields two four-way subcycles and four two-way subcycles.

Size	Glyphs	Comments
2	▢ ▣	Parallelogram. You can get a kinda neat “barber-pole” effect by alternating each period.
4	\\ /-	Approximate, but works even under ASCII.
4	▣▣▣▣	Note that they’re out of order
6	★ ★ ★ ★ ★ ★	Five-spoked asterisk
6	* * * * *	Six-spoked asterisk
5	* * * * *	Fuck your seven-spoked asterisk, fuck expecting six eight-spoked asterisks, and fuck you too.
8	▣ ▣ ▣ ▣ ▣ ▣ ▣ ▣	Some fonts have different sizes. Why? Ugh.
10	▪ □ □ □ □ □ □ □ □ □	The first two are maybe cheating.

TABLE 9: Just a few of Unicode’s many cyclic groups.

Certain transformations can be safely applied to the full Latin alphabet (see Figure 39 for more, some of which I couldn’t realize in $X_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$).

- `w e c a n p u t o u r t e x t i n c i r c l e s`
- `s t r e t t t c h t e x t w i t h f u l l w i d t h`
- `math double struck, for those who wish to look like Rñ`

Others can be performed on a good chunk of Latin script, but are incomplete:

- **PACK MY BOX WITH FIVE DOZEN LIQUOR JUGS** (smallcaps)
- **paCk mY bOx With five dOzen liQuor juGs** (subscripts)
- **Pack my box with five dozen liquor jugs** (superscripts)
- **ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ** (inverted)
- **ƆAƆk mY dox wiTH Fivə bozəni lipUoɹ jUgɹ** (reversed)

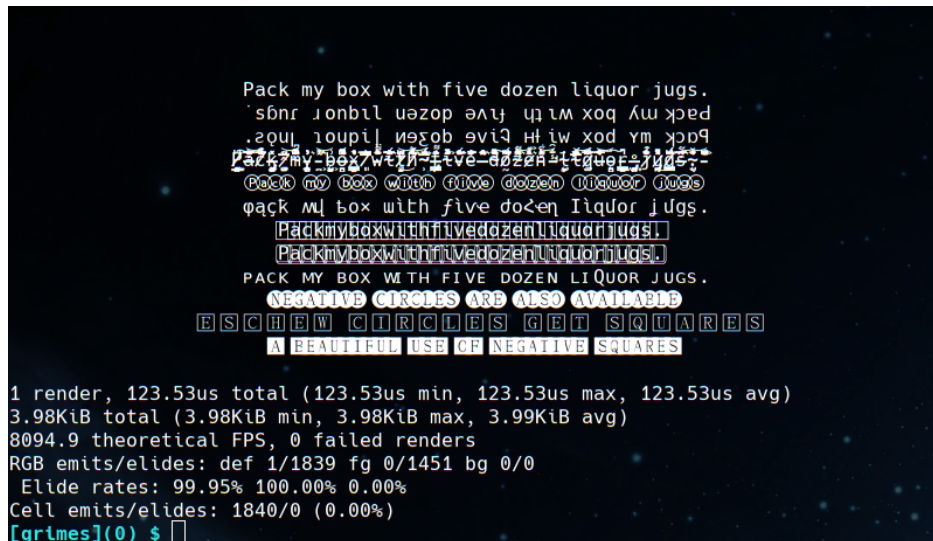


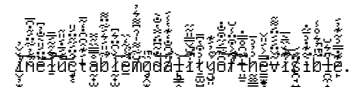
FIGURE 39: A pangram using a variety of Unicode texts.

“Pseudoalphabets” by the thousand exist. These simply substitute glyphs for others of which they are more or less suggestive. There’s no real rhyme or reason behind these pseudoalphabets, but they can be useful if you need a different “font”. Some examples are shown in Figure 40.

A-cute pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
CJK+Thai pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Curvy 1 pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Curvy 2 pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Curvy 3 pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Faux Cyrillic pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Faux Ethiopic pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Math Fraktur pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ
Rock Dots pseudoalphabet	ᄀᄁᄂ ᄃᄄ ᄅᄆ ᄇᄈ ᄉᄊ ᄋᄌ ᄍᄎ ᄏᄐ ᄑᄒ ᄓᄔ ᄕᄖᄗ ᄘᄙ ᄚᄛᄜ

FIGURE 40: Some of “Eli the Bearded”’s pseudoalphabets from <http://qaz.wtf/u/>.

Finally, we can’t speak of stupid Unicode tricks without mentioning *zalgo*:



Dump enough diacritics into an EGC, and the result is a somewhat Cthulhian garble (it’s not quite *squamous*,

and I wouldn't go so far as to call it *eldritch*, but I suppose it's *unheimlich*). Investigating zalgo led down a Reddit hole of madness; I mention it only for completeness, and as an excuse to include Figure 41.

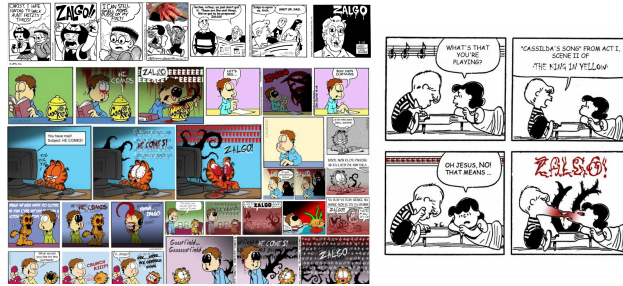


FIGURE 41: Past, present, future, all are one in Yog-Sothoth.

7.7 UTF-8

Unicode Technical Report #17[120] defines seven official Unicode character encoding schemes: UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, and UTF-32LE. What a wealth of encodings! How is one to choose? The -16BE and -16LE forms are simply UTF-16 with a known byte order; a UTF-16 stream can (optionally!) be prefixed with a Byte-Order Mark, at which point the stream reduces to -16LE or -16BE (in the absence of a BOM, the best advice is to follow your heart). UTF-32 breaks down the same way. This question of endianness arises from the fact that UTF-16 and UTF-32 are coded in terms of 16- and 32-bit units. UTF-8, being coded in terms of individual bytes, has no need to define byte order.

“Well, that BOM sounds kinda annoying,” I hear you asking. “What other advantages are offered by UTF-8?” Remember how ANSI X3.4-1986 maps precisely to the first 128 characters of UCS? UTF-8 (and *only* UTF-8, of the official encodings) *encodes* these 128 characters the same as US-ASCII! Boom! Every ASCII document you have—including most source code, configuration files, system files, etc.—is a perfectly valid UTF-8 document. Furthermore, UTF-8 *never encodes non-ASCII characters to the ASCII bytes*. So an arbitrary UTF-8 document may have plenty of high-bit bytes that your ASCII-aware, POSIX-locale program doesn't understand, but it never sees a valid ASCII character where one wasn't intended. UTF-8 encodes ASCII's 0–0x7f to 0–0x7f, and otherwise never produces a byte in that range. This includes the all-important null character 0—Boom! Every nul-terminated C string is a valid UTF-8 string. Every UTF-8 string can be passed through standard C APIs cleanly, and they'll more or less work. It's furthermore self-synchronizing. If you pick up a UTF-8 stream in the middle, you know after reading a single byte whether you're in the middle of a multibyte character.

“Sweet! What's the catch? Does it waste space?” RFC 3629[122] limits UTF-8's range to the $17 * 2^{16}$ -ary code space of UCS, in which case the maximum length of a single UTF-8-encoded UCS code point is four bytes⁶⁴. It's thus always as or more efficient than UCS-32. When the ASCII characters are used, UTF-8 is more efficient than either UTF-16 or UTF-32. Only for streams utterly dominated by BMP codepoints requiring three or more bytes from UTF-8 can UTF-16 encode more efficiently.

“Sweet! What's the catch? Is it super slow?” UTF-32, it is true, allows you to index into a string by character in $O(1)$ (UTF-16 *does not*, unless you're only dealing with BMP strings). UTF-32 also allows you to compute the bytes necessary for encoding in $O(1)$, given the number of Unicode codepoints, but that's only because it's wasteful; if you're willing to be similarly wasteful, you can do the same calculation with UTF-8 (and then trim any wastage at the end, if you wish). Any advantage UTF-32 might hold in lexing simplicity is likely a wash when UTF-8's usual space efficiency is taken into account, owing to more effective use of cache and memory bandwidth. Nope, it's not slow. **Always interoperate in UTF-8 by default.**

⁶⁴You might hear six bytes, and indeed ISO/IEC 10646 specifies six bytes to handle up through U+7FFFFFFF...but only defines UCS to cover 17 planes. Verify your `wctomb(3)` rejects inputs in excess of 0x10ffff before exploiting RFC 3629's tighter bound.

UTF-16 is some truly stupid shit, fit only for jabronies. It only ever passed muster because people thought UCS was going to be a sixteen-bit character set. The moment a second Plane was added, UTF-16 ought have been shown the door. There's an argument to be made for ripping it from the pages of books in your local library. If you must work on a UTF-16 system, use UTF-16 at the boundary, and then keep it around as UTF-32 or UTF-8. Always interoperate—including writing files—in UTF-8 by default.

There are a dozen-odd similarly-named encodings which are useful for nothing but trivia. UCS-2 was UTF-16, but for only the BMP. UCS-4 is just UTF-32. UTF-7 is a seven-bit-clean UTF-8⁶⁵. UTF-1 is UTF-8's older, misshapen sister, locked away from sight in the attic. UTF-5 and UTF-6 were proposed encodings for IDN, but Punycode was selected instead. WTF-8 extends UTF-8 to handle invalid UTF-16 input. BOCU-1 and SCSU are compressing encodings that don't compress as well as gzipped UTF-8. UTF-9 and UTF-18 were jokes. Is UTF-EBCDIC a thing? Of course UTF-EBCDIC is a thing⁶⁶.

The one place where you won't interoperate with UTF-8 is for domain name lookup, when converting IDNA into the LDH subset of ASCII. If you're interested, consult RFC 3492, and Godspeed.

⁶⁵The primary seven-bit-clean media of the modern era is probably email sent without a MIME transfer encoding.

⁶⁶Perhaps the most cursed thing I'm aware of in computing is "UTFE", a variable-length UCS encoding that somehow requires six bytes sometimes, is used exclusively on EBCDIC platforms, and furthermore exclusively only on (drum roll).... **EBCDIC. ORACLE. DATABASES.** Ave Satanas! Also receiving votes: Threaded INTERCAL, non-Postfix mail servers, old-skool XFree86 configuration files with the modeline bullshit, ActiveX controls, "WebNFS", and all Perl ever.

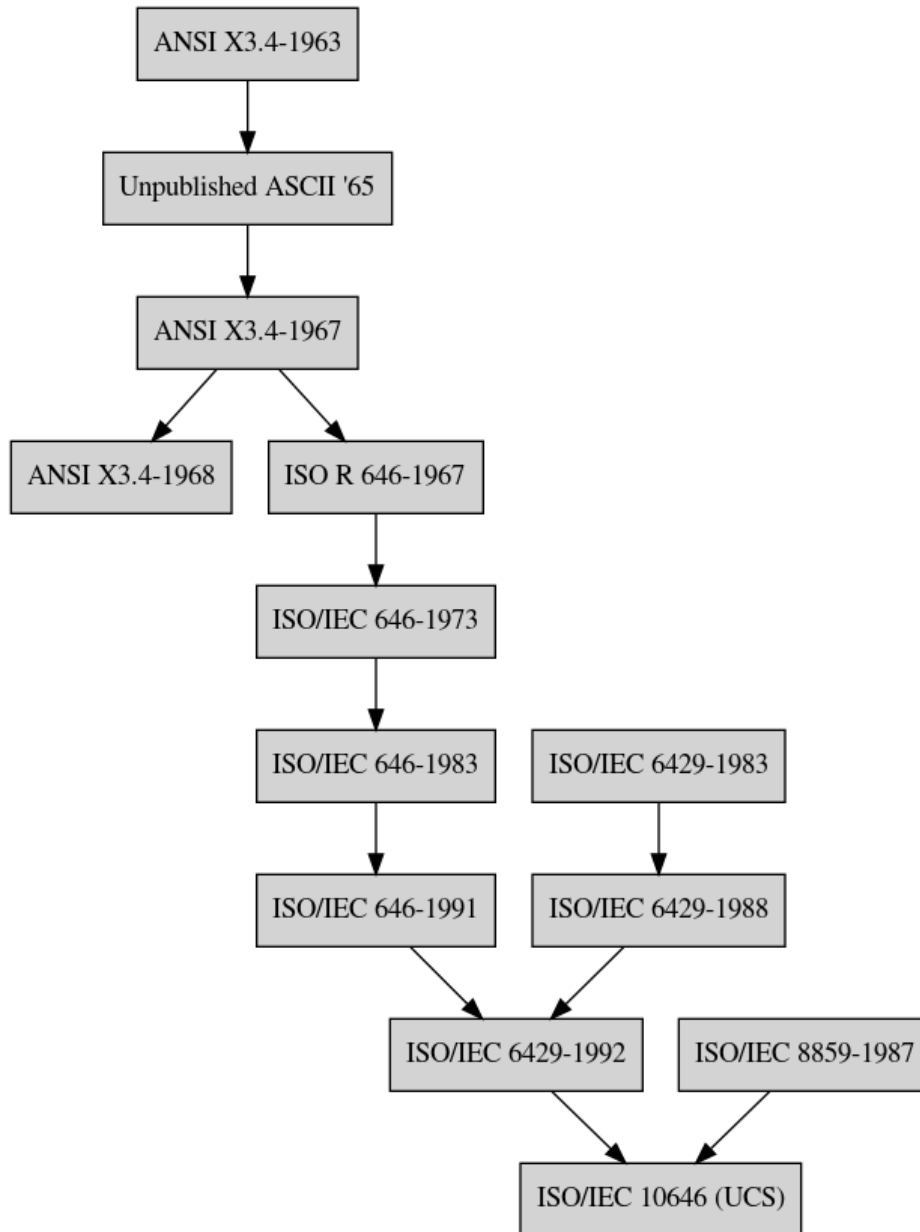


FIGURE 42: Flow of control characters through historic standards.

8 Using ncplanes

Even when I say nothing,
it's a beautiful use of negative space.

Company Flow,
“The Fire in Which you Burn”

As mentioned in Chapter 4, `ncplanes` (henceforth simply `planes`) are the fundamental drawing surface of Notcurses. A Notcurses instance contains a z-axis on which planes are totally ordered⁶⁷. In addition, it always contains at least one plane, the *standard plane*. This plane's origin is always defined to be the rendering area's origin. It is always exactly as large as the rendering area, and it cannot be destroyed.

It's useful to note that there is never a *need* for more than one plane, as demonstrated by the simple fact that each rendered frame is a two-dimensional area. Planes do not add power to the system: *any algorithm which can be expressed using multiple planes can be expressed using a single plane and external state*. Instead, they add expressiveness, and supply order to the state needed beyond the data present in the rendered frame. The color blending performed for transparent or translucent planes can be simulated by the programmer. Redrawing the parts of an underlying plane exposed by moving another can be managed by the programmer. Mapping a base glyph to null cells of different planes can be done by keeping an index for each null cell, etc. etc. In a great many cases, this external structure would be reproducing the algorithms and data structures of planes. **Planes are provided as a concise and efficient implementation of the codes frequently necessary to implement TUIs.** The rule for using planes is thus simply to *use a plane whenever you find yourself implementing code already provided by planes*.

Chapter 4.2 introduced one function that creates a new plane: `ncplane_new()`. In addition, there is `ncplane_aligned()`.

```
// Create a new ncplane at the specified 'yoff', of the specified 'rows' and 'cols'. Align this plane
// according to 'align' relative to 'n'.
struct ncplane* ncplane_aligned(struct ncplane* n, int rows, int cols, int yoff, nalign_e align, void* opaque);
```

LISTING 27: Creating a new plane aligned relative to another.

A plane is defined by:

- A packed “framebuffer” of `cell` structures. Cells are discussed in detail in 8.2; it is enough now to know that each has an EGC, a set of attributes, and a fore- and background color.
- A “base cell”, rendered for any cell with a null EGC.
- A cursor position relative to the plane's origin.
- The plane's position relative to the visible area's origin.
- A two-dimensional size.
- An “egcpool” providing backing storage for the framebuffer's complex EGCs.
- An opaque pointer (the *user pointer*), controlled by the application.
- A current set of attributes and colors.
- A pointer to the plane below this one, or `NULL` for the bottommost plane.

However a plane is created—including the standard plane—it is initialized in the same way. All cells—both the base cell and those of the framebuffer—are zeroed out. A zeroed cell has the null glyph (UTF-8 value “00”), no attributes, and the default foreground and background color (default colors are always opaque). Note that planes are thus by default glyph-transparent but color-opaque. The cursor is placed at the origin.

⁶⁷Future releases of Notcurses might relax this to a partial ordering, allowing multiple `ncplanes` to partition a logical level. See <https://github.com/dankamongmen/notcurses/issues/184>.

The plane's current attributes and channels are likewise zeroed out. The plane is pushed onto the top of the z-axis and assigned an initialized egcpool (see 8.3).

Besides the default plane, planes may occupy any positive size (both the number of rows and columns must be greater than zero), and have their origins any integer offset from the visual origin. It is possible for a plane to be a superset of the visual area, a subset, to exactly match the visual area, to partially overlap it, or even to be entirely off-screen. A plane can be moved to any coordinate, but the plane's cursor cannot be moved off the plane. A plane can be duplicated with `ncplane_dup()`. This will create a new plane of the same

```
// Duplicate an existing ncplane. The new plane will have the same geometry, will duplicate all content, and
// will start with the same rendering state. The new plane will be immediately above the old one on the z axis.
struct ncplane* ncplane_dup(struct ncplane* n, void* opaque);
```

LISTING 28: Duplicating a plane.

geometry at the top of the z-axis. It will then have all other properties duplicated, using its own egcpool. A new user pointer can be provided at duplication time. Each method of creating a plane allows a user

```
void* ncplane_set_userptr(struct ncplane* n, void* opaque);
void* ncplane_userptr(struct ncplane* n);
```

LISTING 29: Manipulating a plane's user pointer.

pointer to be supplied. The plane will never touch this pointer—it exists wholly for the benefit of the calling program. Retrieve it with `ncplane_userptr()`, and reset it with `ncplane_set_userptr()`. The same user pointer can be held by multiple planes, should you so desire.

```
// Destroy the specified ncplane. None of its contents will be visible after the next
// call to notcurses_render(). It is an error to attempt to destroy the standard plane.
int ncplane_destroy(struct ncplane* ncp);

// Destroy all ncplanes other than the standard plane.
void notcurses_drop_planes(struct notcurses* nc);
```

LISTING 30: Destroying planes.

Any plane save the standard plane may be destroyed with `ncplane_destroy()`. All planes save the standard plane may be destroyed in one fell swoop with `notcurses_drop_planes()`.

8.1 Moving and resizing planes

Even the standard plane can be reordered along the z-axis (Listing 31). `ncplane_move_top()` and `ncplane_move_bottom()` are absolute, moving the specified plane to the top or bottom of the z-axis, respectively. `ncplane_move_above()` and `ncplane_move_below()` are relative, moving the plane immediately above or below another one. It is an error to try and move a plane below or above itself, or above or below `NULL`. Likewise, an error will be returned if the relative plane does not exist on the z-axis. All planes other than the standard plane can be moved in the x- and y-dimensions. It is permitted to move a plane partially or even entirely outside of the viewing area. A plane which lies entirely outside the rendering window plays no role in the rendering process. All planes other than the standard plane can be resized (Listing 33). Resizing is a very general and powerful operation—it is possible to implement `ncplane_move_yx()` in terms of `ncplane_resize()`. The four parameters `keepy`, `keepx`, `keepeny`, and `keepenx` define a subset of the plane to retain. The retained rectangle has its origin at `keepy,keepx`, and a `keepeny-row`, `keepenx-column` geometry. If either of the dimensions are zero, no material is retained. In this case, `keepx` and `keepy` are immaterial, save that in no case may any of these four parameters be negative. `keepx` and `keepy` are both relative to the plane's origins, *not* the rendering area. Attempting to “retain” material beyond the boundaries of the plane is an error. `yoff` and `xoff` are likewise relative to the plane's origin, and define the geometry of the plane following the resize. Both of these arguments must be positive. Attempting to retain more material than there is room in the reshaped plane is an error.

```

// Splice ncplane 'n' out of the z-buffer, and reinsert it at the top or bottom.
int ncplane_move_top(struct ncplane* n);
int ncplane_move_bottom(struct ncplane* n);

// Splice ncplane 'n' out of the z-buffer, and reinsert it above/below 'targ'.
int ncplane_move_above_unsafe(struct ncplane* restrict n, struct ncplane* restrict targ);
int ncplane_move_below_unsafe(struct ncplane* restrict n, struct ncplane* restrict targ);

static inline int ncplane_move_above(struct ncplane* n, struct ncplane* above){
    if(n == above){
        return -1;
    }
    return ncplane_move_above_unsafe(n, above);
}

static inline int ncplane_move_below(struct ncplane* n, struct ncplane* below){
    if(n == below){
        return -1;
    }
    return ncplane_move_below_unsafe(n, below);
}

// Return the plane below this one, or NULL if this is at the bottom.
struct ncplane* ncplane_below(struct ncplane* n);

```

LISTING 31: Moving planes on the z axis.

```

// Move this plane relative to the standard plane. It is an error to attempt to move the standard plane.
int ncplane_move_yx(struct ncplane* n, int y, int x);

```

LISTING 32: Moving planes on the x and y axis.

```

int ncplane_resize(struct ncplane* n, int keepy, int keepx, int keepeny,
                  int keepenx, int yoff, int xoff, int ylen, int xlen);

static inline int ncplane_resize_simple(struct ncplane* n, int ylen, int xlen){
    int oldy, oldx;
    ncplane_dim_yx(n, &oldy, &oldx); // current dimensions of 'n'
    int keepeny = oldy > ylen ? ylen : oldy;
    int keepenx = oldx > xlen ? xlen : oldx;
    return ncplane_resize(n, 0, 0, keepeny, keepenx, 0, 0, ylen, xlen);
}

```

LISTING 33: Resizing a plane can retain any amount of the old material.

Yeah, that's a little complex. `ncplane_resize_simple()` is provided as a gentler path to resizing. It retains everything it can (everything, if no shrinking is going on), preferring material towards the upper left. The resulting plane does not move from its origin. Sometimes it is useful to translate coordinates between planes (Listing 34), or between the visible area and planes (this latter is particularly useful when interpreting mouse clicks; see Chapter 12). The content of a plane can be retrieved using reflection (Listing 35). Any coordinate relative to the plane's origin can be provided, along with a value-result `cell` structure. If the coordinates are valid for the plane, the cell at those coordinates will be copied into `c`, and the number of bytes required to store its EGC will be returned. A value less than zero is returned on error.

Just as each cell has a set of attributes and channels, the plane itself has an active attribute set and active channels. These can be freely manipulated using the API of Listings 36, 37, 38, and 39.

```
// provided a coordinate relative to the origin of 'src', map it to the same absolute coordinate
// relative to the origin of 'dst'. either or both of 'y' and 'x' may be NULL.
void ncplane_translate(const struct ncplane* src, const struct ncplane* dst, int* restrict y, int* restrict x);

// Fed absolute 'y'/'x' coordinates, determine whether that coordinate is within the ncplane 'n'. If not, return false.
// If so, return true. Either way, translate the absolute coordinates relative to 'n'. If the point is not within 'n',
// these coordinates will not be within the dimensions of the plane.
bool ncplane_translate_abs(const struct ncplane* n, int* restrict y, int* restrict x);
```

LISTING 34: Translating coordinates between planes.

```
int ncplane_at_cursor(struct ncplane* n, cell* c);
int ncplane_at_yx(struct ncplane* n, int y, int x, cell* c);
```

LISTING 35: Reflecting on the plane to acquire its contents.

```
// get the current channels or attribute word for ncplane 'n'.
uint64_t ncplane_channels(const struct ncplane* n);
uint32_t ncplane_attr(const struct ncplane* n);
```

LISTING 36: Accessing a plane's raw channels and attributes.

During the course of `notcurses_render()`, the plane is examined at all cells intersecting with unsolved coordinates (see 4.3). Whenever the EGC at a cell is the null EGC (a `gcluster` value of 0; recall that this is the default value), the plane's base cell is instead considered for rendering purposes. This applies to glyph, attribute, and colors—there is not yet any means to make multiple cells in a plane glyph-transparent with different colors⁶⁸.

It's possible in rather limited circumstances to perform a rotation of a plane (Listing 42). This only really works for planes entirely populated by half blocks or characters which can be composed from half blocks. The plane is rotated by $\frac{\pi}{2}$ radians in either direction. Note that the rotation is only lossless in terms of color and geometry; it is possible for half blocks to be rotated into full blocks, and vice versa. You are encouraged to consult the source code before making use of rotations.

8.2 Cells

Each coordinate of an plane corresponds to a `cell`. The cell definition is exposed to the application, though it should not generally be directly manipulated. A multicolumn cell (a cell containing an EGC of n columns where $n > 1$) overrides the $n - 1$ following cells. Since there are always a fixed number of cells, this means that the overridden cells are skipped during rendering, as well as being zeroed out at the time the multicolumn EGC is written to the cell. The `gcluster` field is a 32-bit number. If the value is less than 128, it directly specifies its UTF-8 encoded character. Since Unicode's first 128 values are taken directly from ASCII, this means the entirety of ASCII can be represented in-line. If the value is greater than or equal to 128, it is a bias-128 index into the plane's associated `egcpool`. Since `egcpools` are per-plane, this implies that it is unsafe to blindly copy a cell from one plane to another.

Applications generally need not work directly with cells, though sometimes it is easiest to do so. The usual reason for working with a cell is either to set all three properties of output at once (glyph, attributes, and colors), or to receive all three properties at once when retrieving a coordinate's data.

As further discussed in Chapter 9.3, the `channels` variable is a 64-bit field packing together a number of properties. The high 32 bits apply to the foreground, and the low 32 bits to the background. They can be set and queried as a channel (Listing 44).

RGB values consume 24 bits of each channel, 75% of the 64 bits in `channels`. RGB values can be blended, clipped, and otherwise dealt with arithmetically.

⁶⁸See <https://github.com/dankamongmen/notcurses/issues/395>.

```

// Extract the 32-bit working foreground channel from an ncplane.
static inline unsigned ncplane_fchannel(const struct ncplane* nc){
    return channels_fchannel(ncplane_channels(nc));
}

// Extract 24 bits of working foreground RGB from an ncplane, shifted to LSBs.
static inline unsigned ncplane_fg(const struct ncplane* nc){
    return channels_fg(ncplane_channels(nc));
}

// Extract 2 bits of foreground alpha from 'struct ncplane', shifted to LSBs.
static inline unsigned ncplane_fg_alpha(const struct ncplane* nc){
    return channels_fg_alpha(ncplane_channels(nc));
}

// Extract 24 bits of foreground RGB from 'n', split into subcomponents.
static inline unsigned ncplane_fg_rgb(const struct ncplane* n, unsigned* r, unsigned* g, unsigned* b){
    return channels_fg_rgb(ncplane_channels(n), r, g, b);
}

// Set the current fore/background color using RGB specifications. If the terminal does not support
// directly-specified 3x8b cells (24-bit "TrueColor", indicated by the "RGB" terminfo capability),
// the provided values will be interpreted in some lossy fashion. None of r, g, or b may exceed 255.
// "HP-like" terminals require setting foreground and background at the same time using "color pairs";
// notcurses will manage color pairs transparently.
int ncplane_set_fg_rgb(struct ncplane* n, int r, int g, int b);

// Same, but clipped to [0..255].
void ncplane_set_fg_rgb_clipped(struct ncplane* n, int r, int g, int b);

// Same, but with rgb assembled into a channel (i.e. lower 24 bits).
int ncplane_set_fg(struct ncplane* n, unsigned channel);

// Use the default color for the foreground/background.
void ncplane_set_fg_default(struct ncplane* n);

// Set the ncplane's foreground palette index, set the foreground palette index bit, set it
// foreground-opaque, and clear the foreground default color bit.
int ncplane_set_fg_palindex(struct ncplane* n, int idx);

// Set the alpha parameters for ncplane 'n'.
int ncplane_set_fg_alpha(struct ncplane* n, int alpha);

```

LISTING 37: Manipulating a plane's active foreground channel.

It is also possible to make use of palette-indexed color (recall that the size of the palette can be acquired with `notcurses_palette_size()`). Palette-indexed color requires much less bandwidth than pure RGB (indeed, work is underway to emit palette-indexed rasterization even when RGB has been provided—see <https://github.com/dankamongmen/notcurses/issues/371>), and allows for finer control on terminals which don't faithfully implement RGB TrueColor. The terminal palette can be manually reprogrammed with the `palette256` API (see Listing 49).

Finally, the default fore- and/or background color can be used, and is indeed the default. Default colors can't be blended. Some terminals can be configured to use a transparent background. Only in cells using

```

// Extract the 32-bit working background channel from an ncplane.
static inline unsigned ncplane_bchannel(const struct ncplane* nc){
    return channels_bchannel(ncplane_channels(nc));
}

// Extract 24 bits of working background RGB from an ncplane, shifted to LSBs.
static inline unsigned ncplane_bg(const struct ncplane* nc){
    return channels_bg(ncplane_channels(nc));
}

// Extract 2 bits of background alpha from 'struct ncplane', shifted to LSBs.
static inline unsigned ncplane_bg_alpha(const struct ncplane* nc){
    return channels_bg_alpha(ncplane_channels(nc));
}

// Extract 24 bits of background RGB from 'n', split into subcomponents.
static inline unsigned ncplane_bg_rgb(const struct ncplane* n, unsigned* r, unsigned* g, unsigned* b){
    return channels_bg_rgb(ncplane_channels(n), r, g, b);
}

int ncplane_set_bg_rgb(struct ncplane* n, int r, int g, int b);
void ncplane_set_bg_rgb_clipped(struct ncplane* n, int r, int g, int b);
int ncplane_set_bg(struct ncplane* n, unsigned channel);
void ncplane_set_bg_default(struct ncplane* n);
int ncplane_set_bg_palindex(struct ncplane* n, int idx);
int ncplane_set_bg_alpha(struct ncplane* n, int alpha);

```

LISTING 38: Manipulating a plane's active background channel.

```

// Set the specified style bits for the ncplane 'n', whether they're actively supported or not.
void ncplane_styles_set(struct ncplane* n, unsigned stylebits);

// Add the specified styles to the ncplane's existing spec.
void ncplane_styles_on(struct ncplane* n, unsigned stylebits);

// Remove the specified styles from the ncplane's existing spec.
void ncplane_styles_off(struct ncplane* n, unsigned stylebits);

// Return the current styling for this ncplane.
unsigned ncplane_styles(const struct ncplane* n);

```

LISTING 39: Manipulating a plane's active attributes.

the default background color can this effect be seen.

8.3 egcpools

Each plane is backed by an `egcpool` structure. Any cell requiring more than a single byte to encode its EGC will write the EGC to the `egcpool`, and store a byte-granular index into the actual `gcluster` field of the `cell`. The first 128 UTF-8 characters are stored directly, and thus any `gcluster` value greater than or equal to 128 is actually a biased⁶⁹ index into the pool. The pool can grow as large as 2^{25} bytes (32 MiB). Each EGC is stored as a NUL-terminated string, so the minimum size is three bytes (since all single-byte UTF-8 is inlined, the minimum UTF-8 EGC to be placed in the pool is two bytes, plus a NUL byte).

⁶⁹128-biased, i.e. `gcluster` is the offset + 128.


```

// Set the ncplane's base cell to this cell. It will be used for purposes of rendering anywhere that the ncplane's
// gcluster is 0. Erasing the ncplane does not reset the base cell; this function must be called with a zero 'c'.
int ncplane_set_base_cell(struct ncplane* ncp, const cell* c);

// Set the ncplane's base cell to this cell. It will be used for purposes of rendering anywhere that the ncplane's
// gcluster is 0. Erasing the ncplane does not reset the base cell; this function must be called with an empty
// 'egc'. 'egc' must be a single extended grapheme cluster.
int ncplane_set_base(struct ncplane* ncp, uint64_t channels, uint32_t attrword, const char* egc);

// Extract the ncplane's base cell into 'c'. The reference is invalidated if 'ncp' is destroyed.
int ncplane_base(struct ncplane* ncp, cell* c);

```

LISTING 40: Manipulating a plane's base cell.

```

// Alignment within the ncplane. Left/right-justified, or centered.
typedef enum { NCALIGN_LEFT, NCALIGN_CENTER, NCALIGN_RIGHT, } ncalign_e;

// Return the column at which 'c' cols ought start in order to be aligned according to 'align' within ncplane 'n'.
// Returns INT_MAX on invalid 'align'. Undefined behavior on negative 'c'.
static inline int ncplane_align(const struct ncplane* n, ncalign_e align, int c){
    if(align == NCALIGN_LEFT){
        return 0;
    }
    int cols = ncplane_dim_x(n);
    if(align == NCALIGN_CENTER){
        return (cols - c) / 2;
    }else if(align == NCALIGN_RIGHT){
        return cols - c;
    }
    return INT_MAX;
}

```

LISTING 41: Aligning output within a plane.

```

int ncplane_rotate_cw(struct ncplane* n);
int ncplane_rotate_ccw(struct ncplane* n);

```

LISTING 42: Rotating planes.

```

typedef struct cell {
    // These 32 bits are either a single-byte, single-character grapheme cluster (values 0-0x7f), or
    // an offset into a per-ncplane attached pool of varying-length UTF-8 grapheme clusters.
    uint32_t gcluster;        // 4B -> 4B
    uint32_t attrword;       // + 4B -> 8B
    uint64_t channels;       // + 8B == 16B
} cell;

```

LISTING 43: The `cell` definition.

You shouldn't ever need to work directly with `egcpools`, and their API is not exposed. It is possible to exhaust an `egcpool`, but not without either a tremendous geometry (start worrying around 1024x1024 or so), or a plenitude of pathological EGCs. Should this unhappy situation occur, functions like `ncplane_putegc()` and even `cell_load()` will start failing (they should not crash or otherwise fault). There's not much you can do at this point save erase or destroy the plane⁷⁰. Outside of deliberate attempts to trigger it during testing, I

⁷⁰This ought be addressed by <https://github.com/dankamongmen/notcurses/issues/425>.

```

// Extract the 32-bit background channel from a cell.
static inline unsigned cell_bchannel(const cell* c1){
    return channels_bchannel(c1->channels);
}

// Extract the 32-bit foreground channel from a cell.
static inline unsigned cell_fchannel(const cell* c1){
    return channels_fchannel(c1->channels);
}

// Set the 32-bit background channel of a cell.
static inline uint64_t cell_set_bchannel(cell* c1, uint32_t channel){
    return channels_set_bchannel(&c1->channels, channel);
}

// Set the 32-bit foreground channel of a cell.
static inline uint64_t cell_set_fchannel(cell* c1, uint32_t channel){
    return channels_set_fchannel(&c1->channels, channel);
}

```

LISTING 44: Modifying `cell` channels.

have never seen this failure case.

8.4 Alpha blending and plane transparency

The rendering algorithm described in Chapter 4.3 is responsible for the blending of colors and selection of glyphs from among planes. Let’s look at it in full detail.

There are three largely independent dimensions for each rendered cell, and they are the same three dimensions of any `cell`: EGC plus attribute, foreground color, and background color. There are three different ways that colors can be set in the actual terminal (assuming support):

- The “default foreground“ and “default background“ colors. These are inherited from the terminal, where they are usually user-configurable. Since most of the user’s shell experience will take place in these colors, you can safely assume that either there is a high contrast difference between the two, or the user doesn’t care much about contrast. Other than that, it is not safe to assume that the default background is black, white, blue, or even monochromatic—most terminal emulators allow an image to be set as the background (in which case the “default background“ color can be considered translucent atop this opaque background), or even a (perhaps partially) transparent background (the “default background“ color is translucent atop the composited desktop)⁷¹. The “op” terminfo capability resets both channels, requiring the other channel (assuming it to *not* be the default) to be emitted even if it would normally have been elided.
- Palette-indexed color. The size of the palette (usually 1, 2, 8, 16, 88, or 256) is indicated by the `colors` terminfo capability⁷². If the `ccc` or `initp` terminfo capabilities are present, the palette can be modified.
- RGB color. 24 bits as 3 channels of 8 bits each are directly specified. Support is indicated by the terminfo `rgb` capability, or by a user-supplied `COLORTERM` environment variable having the value “24bit” or “truecolor”.

Currently, only RGB can be blended. When rendering, recall that a color dimension is “solved” when the computed cell reaches `CELL_ALPHA_OPAQUE` in that channel. The channel is initialized to `CELL_ALPHA_TRANSPARENT`. What is done at each intersecting cell depends on the computed alpha thus far; see Table 10.

⁷¹I can only speak for myself, as a Solarized-on-black enthusiast, but a program which forces a largely white background on

```

// do not pass palette-indexed channels!
static inline uint64_t cell_blend_fchannel(cell* c1, unsigned channel, unsigned* blends){
    return cell_set_fchannel(c1, channels_blend(cell_fchannel(c1), channel, blends));
}

// Extract 24 bits of foreground RGB from 'cell', shifted to LSBs.
static inline unsigned cell_fg(const cell* c1){
    return channels_fg(c1->channels);
}

// Extract 2 bits of foreground alpha from 'cell', shifted to LSBs.
static inline unsigned cell_fg_alpha(const cell* c1){
    return channels_fg_alpha(c1->channels);
}

// Extract 24 bits of foreground RGB from 'cell', split into components.
static inline unsigned cell_fg_rgb(const cell* c1, unsigned* r, unsigned* g, unsigned* b){
    return channels_fg_rgb(c1->channels, r, g, b);
}

// Set the r, g, and b cell for the foreground component of this 64-bit
// 'cell' variable, and mark it as not using the default color.
static inline int cell_set_fg_rgb(cell* c1, int r, int g, int b){
    return channels_set_fg_rgb(&c1->channels, r, g, b);
}

// Same, but clipped to [0..255].
static inline void cell_set_fg_rgb_clipped(cell* c1, int r, int g, int b){
    channels_set_fg_rgb_clipped(&c1->channels, r, g, b);
}

// Same, but with an assembled 24-bit RGB value.
static inline int cell_set_fg(cell* c, uint32_t channel){
    return channels_set_fg(&c->channels, channel);
}

```

LISTING 45: cell foreground RGBA functionality.

8.5 Manual palette-indexed color

While it shouldn't ever be necessary, some algorithms are more easily expressed using palette-indexed color. Functions for manually manipulating the palette are available when supported by the terminal (as advertised by the “ccc” terminfo capability). The palette is simply an integer-indexed set of RGB values, where those RGB values are maintained by the terminal. It is possible to change them, in which case any output expressed with that palette index will be updated to reflect the change. The API for manipulating the palette is provided in Listing 49.

8.6 Fading and pulsing planes

When we speak of palettes, one thing comes to mind: blingful palette fades. Back in my misspent adolescence, this was basically the “Hello World” of x86 assembly. Whip it into MCGA mode 13h, load up 0xa000, and drive those 64KB. But I reminisce...fades are available at the plane level, as is pulsing (a periodic fade); see

a fullscreen application annoys the hell out of me. Try to respect the user's configured background where possible.

⁷²A bad TERM setting will wreck havoc on colors.

```

static inline uint64_t cell_blend_bchannel(cell* c1, unsigned channel, unsigned* blends){
    return cell_set_bchannel(c1, channels_blend(cell_bchannel(c1), channel, blends));
}

// Extract 24 bits of background RGB from 'cell', shifted to LSBs.
static inline unsigned cell_bg(const cell* c1){
    return channels_bg(c1->channels);
}

// Extract 2 bits of background alpha from 'cell', shifted to LSBs.
static inline unsigned cell_bg_alpha(const cell* c1){
    return channels_bg_alpha(c1->channels);
}

// Extract 24 bits of background RGB from 'cell', split into components.
static inline unsigned cell_bg_rgb(const cell* c1, unsigned* r, unsigned* g, unsigned* b){
    return channels_bg_rgb(c1->channels, r, g, b);
}

// Set the r, g, and b cell for the background component of this 64-bit
// 'cell' variable, and mark it as not using the default color.
static inline int cell_set_bg_rgb(cell* c1, int r, int g, int b){
    return channels_set_bg_rgb(&c1->channels, r, g, b);
}

// Same, but clipped to [0..255].
static inline void cell_set_bg_rgb_clipped(cell* c1, int r, int g, int b){
    channels_set_bg_rgb_clipped(&c1->channels, r, g, b);
}

// Same, but with an assembled 24-bit RGB value.
static inline int cell_set_bg(cell* c, uint32_t channel){
    return channels_set_bg(&c->channels, channel);
}

```

LISTING 46: cell background RGBA functionality.

Listing 50. Ironically, this only works for RGB data at the moment, but that is a temporary restriction.

The pulsing and fading functions all accept an optional callback of type `fadecb` (Listing 51). If `NULL` is provided as the callback, the functions will call `notcurses_render()` in the callback's place. Otherwise, the callback is provided the root `notcurses` struct, the plane of operation, and a user-provided, per-operation curry. If the callback does not itself call `notcurses_render()`, this frame of the fade will not be rendered.

```
// Set the cell's foreground palette index, set the foreground palette index
// bit, set it foreground-opaque, and clear the foreground default color bit.
static inline int cell_set_fg_palindex(cell* c1, int idx){
    if(idx < 0 || idx >= NCPALETTESIZE){
        return -1;
    }
    c1->channels |= CELL_FGDEFAULT_MASK;
    c1->channels |= CELL_FG_PALETTE;
    c1->channels &= ~(CELL_ALPHA_MASK << 32u);
    c1->attrword &= 0xffff00ff;
    c1->attrword |= (idx << 8u);
    return 0;
}

static inline unsigned cell_fg_palindex(const cell* c1){
    return (c1->attrword & 0x0000ff00) >> 8u;
}

// Set the cell's background palette index, set the background palette index
// bit, set it background-opaque, and clear the background default color bit.
static inline int cell_set_bg_palindex(cell* c1, int idx){
    if(idx < 0 || idx >= NCPALETTESIZE){
        return -1;
    }
    c1->channels |= CELL_BGDEFAULT_MASK;
    c1->channels |= CELL_BG_PALETTE;
    c1->channels &= ~CELL_ALPHA_MASK;
    c1->attrword &= 0xffffffff00;
    c1->attrword |= idx;
    return 0;
}

static inline unsigned cell_bg_palindex(const cell* c1){
    return c1->attrword & 0x000000ff;
}

static inline bool cell_fg_palindex_p(const cell* c1){
    return channels_fg_palindex_p(c1->channels);
}

static inline bool cell_bg_palindex_p(const cell* c1){
    return channels_bg_palindex_p(c1->channels);
}
```

LISTING 47: cell palette-indexed color functionality.

```

// Is the background using the "default background color"? The "default background color"
// must generally be used to take advantage of terminal-effected transparency.
static inline bool cell_bg_default_p(const cell* c1){
    return channels_bg_default_p(c1->channels);
}

// Is the foreground using the "default foreground color"?
static inline bool cell_fg_default_p(const cell* c1){
    return channels_fg_default_p(c1->channels);
}

```

LISTING 48: cell default color functionality.

Alpha / Blendcount	Mode	In-alpha	In-mode	Result
CELL_ALPHA_TRANSPARENT/0	any	CELL_ALPHA_TRANSPARENT	any	No change
CELL_ALPHA_TRANSPARENT/0	any	CELL_ALPHA_BLEND or CELL_ALPHA_OPAQUE	any	State ← Incoming Blendcount ← 1
CELL_ALPHA_BLEND/n	RGB	CELL_ALPHA_TRANSPARENT	any	No change
CELL_ALPHA_BLEND/n	RGB	any	Default/palette	No change
CELL_ALPHA_BLEND/n	RGB	CELL_ALPHA_BLEND	RGB	Blend ++Blendcount
CELL_ALPHA_BLEND/n	RGB	CELL_ALPHA_OPAQUE	RGB	Blend mode ← OPAQUE ++Blendcount
CELL_ALPHA_OPAQUE/n	any	any	any	No change

TABLE 10: Transition matrix for alpha blending. On the foreground channel, encountering CELL_ALPHA_HIGHCONTRAST sets a bit, and then behaves as if it had been CELL_ALPHA_OPAQUE. Upon solving the cell, if the highcontrast bit is set, either white or black is blended into the foreground as necessary until a minimum contrast level is reached vis-à-vis the background.

```
typedef struct palette256 {
    uint32_t chans[NCPALETTE_SIZE]; // We store the RGB values as a regular ol' channel
} palette256;

// Create a new palette store. It will be initialized with notcurses's best
// knowledge of the currently configured palette.
palette256* palette256_new(struct notcurses* nc);

// Attempt to configure the terminal with the provided palette 'p'. Does not
// transfer ownership of 'p'; palette256_free() can still be called.
int palette256_use(struct notcurses* nc, const palette256* p);

// Manipulate entries in the palette store 'p'. These are *not* locked.
static inline int palette256_set_rgb(palette256* p, int idx, int r, int g, int b){
    if(idx < 0 || (size_t)idx > sizeof(p->chans) / sizeof(*p->chans)){
        return -1;
    }
    return channel_set_rgb(&p->chans[idx], r, g, b);
}

static inline int palette256_set(palette256* p, int idx, unsigned rgb){
    if(idx < 0 || (size_t)idx > sizeof(p->chans) / sizeof(*p->chans)){
        return -1;
    }
    return channel_set(&p->chans[idx], rgb);
}

static inline int palette256_get_rgb(const palette256* p, int idx, unsigned* restrict r,
                                   unsigned* restrict g, unsigned* restrict b);
    if(idx < 0 || (size_t)idx > sizeof(p->chans) / sizeof(*p->chans)){
        return -1;
    }
    return channel_rgb(p->chans[idx], r, g, b);
}

// Free the palette store 'p'.
void palette256_free(palette256* p);

// Convert the plane's content to greyscale.
void ncplane_greyscale(struct ncplane* n);
```

LISTING 49: The palette256 API facilitates manual palette programming.

```

// Fade the ncplane out over the provided time, calling the specified function when done. Requires a terminal
// which supports truecolor, or at least palette modification (if the terminal uses a palette, our ability to
// fade planes is limited, and affected by the complexity of the rest of the screen). It is not safe to
// resize or destroy the plane during the fadeout.
int ncplane_fadeout(struct ncplane* n, const struct timespec* ts, fade_cb fader, void* curry);

// Fade the ncplane in over the specified time. Load the ncplane with the target cells without rendering, then
// call this function. When it's done, the ncplane will have reached the target levels, starting from zeroes.
int ncplane_fadein(struct ncplane* n, const struct timespec* ts, fade_cb fader, void* curry);

// Pulse the plane in and out until the callback returns non-zero, relying on the callback 'fader' to initiate
// rendering. 'ts' defines the half-period (i.e. the transition from black to full brightness, or back again).
// Proper use involves preparing (but not rendering) an ncplane, then calling ncplane_pulse(), which will fade
// in from black to the specified colors.
int ncplane_pulse(struct ncplane* n, const struct timespec* ts, fade_cb fader, void* curry);

```

LISTING 50: Palette fades.

```

// Called for each delta performed in a fade on ncp. If anything but 0 is returned, the fading operation ceases
// immediately, and that value is propagated out. If provided and not NULL, the faders will not themselves call
// notcurses_render().
typedef int (*fade_cb)(struct notcurses* nc, struct ncplane* ncp, void* curry);

```

LISTING 51: Callback type for pulsing and fading.

9 Writing and styling text

```
My pad and my pen
(ah ah, you didn't go there)!
```

A Tribe Called Quest,
"Pad and Pen"

The most fundamental aspect of textual interfaces is, after all, text. All valid UTF-8 can be written to a plane. If scrolling has been enabled for a plane, any amount of text can be written. If scrolling has not been enabled, output can only be generated through the end of the current line. All text output functions return the number of screen columns written as their primary return value (or a negative number on failure), and load the number of bytes written into an auxiliary value-result parameter. If the supplied output would exceed the line, a short number of columns will be returned. Scrolling is disabled by default on the standard plane, and on all new planes.

```
// Move the cursor to the specified position (the cursor needn't be visible). Returns -1 on error,
// including negative parameters, or ones exceeding the plane's dimensions.
int ncplane_cursor_move_yx(struct ncplane* n, int y, int x);

// Get the current position of the cursor within n. y and/or x may be NULL.
void ncplane_cursor_yx(const struct ncplane* n, int* restrict y, int* restrict x);
```

LISTING 52: Cursor management. Each plane has its own cursor.

The cursor is advanced appropriately to the cell just beyond the output. If the entire line was written, and scrolling is not enabled, the cursor will be off-plane and must be repositioned before writing any further. If scrolling is enabled, the cursor will either move to the first column of the next line, or if the cursor is on the last line, the plane will be scrolled by one line (and the cursor will move to the first column). If there is an error following some output, the cursor will be positioned following the generated output. Comparing the new cursor location to the old can thus reveal the amount of output generated in the event of a failure.

To check that there was no failure, then, verify that the return value is not negative. The entirety of the input was written if any of the following is true:

- The return value is equal to the number of columns required to represent the input. `mbswidth()` (Listing 53) can be used to find the number of columns required by the input.
- The number of bytes written is equal to the number of bytes supplied. Since Notcurses supports only UTF-8 and ASCII, `strlen()` can be used to find the number of bytes supplied.
- The cursor is not beyond the end of the plane, and scrolling is disabled on the plane.

```
// Calculate the size in columns of the provided UTF8 multibyte string.
int mbswidth(const char* mbs);
```

LISTING 53: `mbswidth()` counts columns in a multibyte string.

9.1 Writing text to planes

Multiple families of functions are available for writing to planes. Only valid encoded sequences from the active locale's encoding can be output. Attempting to e.g. write UTF-8 characters while using the `ANSI_X3.4-1968` encoding will fail as soon as a non-ASCII (multibyte) character is submitted.

The base form of each family places its output at the plane's current cursor location. Each family has a `_xy()`-suffixed form which moves the cursor as specified prior to beginning output. Supplying `-1` to the `x` and `y` parameters of these forms doesn't move the cursor on the relevant axis. Supplying `-1` to both decays to the base function of the family. The `_stainable()`-suffixed form updates the glyphs of a plane without changing the attributes or channels.

```
// Replace the cell at the specified coordinates with the provided cell 'c', and advance the cursor by
// the width of the cell (but not past the end of the plane). On success, returns the number of columns
// the cursor was advanced. On failure, -1 is returned.
int ncplane_putc_yx(struct ncplane* n, int y, int x, const cell* c);

// Call ncplane_putc_yx() for the current cursor location.
static inline int ncplane_putc(struct ncplane* n, const cell* c){
    return ncplane_putc_yx(n, -1, -1, c);
}
```

LISTING 54: Output of cells to planes.

ASCII (and thus the lowest 128 UTF-8 encoded characters) can be written directly with the `putsimple` family. Note that any control character will be replaced with a space.

```
// Replace the EGC underneath us with that of 'c', but retain the styling. The current styling of the
// plane will not be changed. Replace the cell at the specified coordinates with the provided 7-bit char
// 'c'. Advance the cursor by 1. On success, returns 1. On failure, returns -1. This works whether the
// underlying char is signed or unsigned.
int ncplane_putsimple_yx(struct ncplane* n, int y, int x, char c);

// Call ncplane_putsimple_yx() at the current cursor location.
static inline int
ncplane_putsimple(struct ncplane* n, char c){
    return ncplane_putsimple_yx(n, -1, -1, c);
}

// Replace the EGC underneath us, but retain the styling. The current styling
// of the plane will not be changed.
int ncplane_putsimple_stainable(struct ncplane* n, char c);
```

LISTING 55: Direct output of single-byte UTF-8 to planes.

On systems where `wchar_t` is at least twenty-five bits⁷³, a single `wchar_t` can represent any UCS code point (though not necessarily any EGC). The `putwc` family (Listing 56) allows direct output of a single `wchar_t`.

```
// Replace the cell at the specified coordinates with the provided wide char 'w'. Advance the cursor
// by the character's width as reported by wcwidth(). On success, returns 1. On failure, returns -1.
static inline int ncplane_putwc_yx(struct ncplane* n, int y, int x, wchar_t w){
    wchar_t warr[2] = { w, L'\0' };
    return ncplane_putwstr_yx(n, y, x, warr);
}

// Call ncplane_putwc() at the current cursor position.
static inline int ncplane_putwc(struct ncplane* n, wchar_t w){
    return ncplane_putwc_yx(n, -1, -1, w);
}
```

LISTING 56: Direct output of a single `wchar_t`.

EGCs can be output one at a time. Supplying multiple EGCs in a single buffer to the `putegc` family (Listing 57) will only ever see the first one output.

⁷³`wchar_t` is only 16 bits on some systems, usually those same brain-damaged ones which make great use of brain-damaged UTF-16. Such a `wchar_t` can only represent characters from the Basic Multilingual Plane; the other sixteen planes require use of *surrogate characters*.

```

// Replace the cell at the specified coordinates with the provided EGC, and advance the cursor by the
// width of the cluster (but not past the end of the plane). On success, returns the number of columns
// the cursor was advanced. On failure, -1 is returned. The number of bytes converted from gclust is
// written to 'sbytes' if non-NULL.
int ncplane_putegc_yx(struct ncplane* n, int y, int x, const char* gclust, int* sbytes);

// Call ncplane_putegc() at the current cursor location.
static inline int ncplane_putegc(struct ncplane* n, const char* gclust, int* sbytes){
    return ncplane_putegc_yx(n, -1, -1, gclust, sbytes);
}

// Replace the EGC underneath us, but retain the styling. The current styling
// of the plane will not be changed.
int ncplane_putegc_stainable(struct ncplane* n, const char* gclust, int* sbytes);

```

LISTING 57: Output of single EGCs to planes.

```

#define WCHAR_MAX_UTF8BYTES 6

// ncplane_putegc(), but following a conversion from wchar_t to UTF-8 multibyte.
static inline int ncplane_putwegc(struct ncplane* n, const wchar_t* gclust, int* sbytes){
    // maximum of six UTF8-encoded bytes per wchar_t
    const size_t mbytes = (wcslen(gclust) * WCHAR_MAX_UTF8BYTES) + 1;
    char* mbstr = (char*)malloc(mbytes); // need cast for c++ callers
    if(mbstr == NULL){
        return -1;
    }
    size_t s = wcstombs(mbstr, gclust, mbytes);
    if(s == (size_t)-1){
        free(mbstr);
        return -1;
    }
    int ret = ncplane_putegc(n, mbstr, sbytes);
    free(mbstr);
    return ret;
}

// Call ncplane_putwegc() after successfully moving to y, x.
static inline int ncplane_putwegc_yx(struct ncplane* n, int y, int x, const wchar_t* gclust, int* sbytes){
    if(ncplane_cursor_move_yx(n, y, x)){
        return -1;
    }
    return ncplane_putwegc(n, gclust, sbytes);
}

// Replace the EGC underneath us, but retain the styling. The current styling
// of the plane will not be changed.
int ncplane_putwegc_stainable(struct ncplane* n, const wchar_t* gclust, int* sbytes);

```

LISTING 58: Output of single wchar_t-encoded EGCs to planes.

Finally, analogues of `printf(3)` and `vprintf(3)` are provided (Listing 61).

```

// Write a series of EGCs to the current location, using the current style. They will be interpreted as a
// series of columns (according to the definition of ncplane_putc()). Advances the cursor by some positive
// number of cells (though not beyond the end of the plane); this number is returned on success. On error,
// a non-positive number is returned, indicating the number of cells which were written before the error.
int ncplane_putstr_yx(struct ncplane* n, int y, int x, const char* gclustarr);

static inline int
ncplane_putstr(struct ncplane* n, const char* gclustarr){
    return ncplane_putstr_yx(n, -1, -1, gclustarr);
}

int ncplane_putstr_aligned(struct ncplane* n, int y, nalign_e align, const char* s);

```

LISTING 59: Output of strings to planes.

```

// ncplane_putstr(), but following a conversion from wchar_t to UTF-8 multibyte.
static inline int ncplane_putwstr_yx(struct ncplane* n, int y, int x, const wchar_t* gclustarr){
    // maximum of six UTF8-encoded bytes per wchar_t
    const size_t mbytes = (wcslen(gclustarr) * WCHAR_MAX_UTF8BYTES) + 1;
    char* mbstr = (char*)malloc(mbytes); // need cast for c++ callers
    if(mbstr == NULL){
        return -1;
    }
    size_t s = wcstombs(mbstr, gclustarr, mbytes);
    if(s == (size_t)-1){
        free(mbstr);
        return -1;
    }
    int ret = ncplane_putstr_yx(n, y, x, mbstr);
    free(mbstr);
    return ret;
}

static inline int ncplane_putwstr_aligned(struct ncplane* n, int y, nalign_e align, const wchar_t* gclustarr){
    int width = wcswidth(gclustarr, INT_MAX);
    int xpos = ncplane_align(n, align, width);
    return ncplane_putwstr_yx(n, y, xpos, gclustarr);
}

static inline int ncplane_putwstr(struct ncplane* n, const wchar_t* gclustarr){
    return ncplane_putwstr_yx(n, -1, -1, gclustarr);
}

```

LISTING 60: Output of wide strings to planes.

9.2 The 32-bit attribute value

Each cell has a 32-bit attribute field, initialized to 0. Many functions accept a 32-bit attribute, either to directly set a cell, or to apply to a number of cells. This type (sometimes called `attrword`) is logically a 2-byte bitmask of `NCSTYLE_` flags (see Table 11), followed by two 8-bit palette indices. The palette indices may take any value from 0–255, but are used if and only if the corresponding “not default color” and “palette-indexed color” bits are set in the channels (see below). Changing a channel to indicate default color will *not* reset the palette byte, but *will* supersede it⁷⁴.

⁷⁴Applications are strongly encouraged not to treat these two bytes as free-use scratchpad, in case these semantics change in the future. With that said, sometimes you’ve gotta take sixteen bits wherever you can find them, especially when they represent

```

// The ncplane equivalents of printf(3) and vprintf(3).
int ncplane_vprintf_aligned(struct ncplane* n, int y, ncalign_e align, const char* format, va_list ap);

int ncplane_vprintf_yx(struct ncplane* n, int y, int x, const char* format, va_list ap);

static inline int ncplane_vprintf(struct ncplane* n, const char* format, va_list ap){
    return ncplane_vprintf_yx(n, -1, -1, format, ap);
}

static inline int ncplane_printf(struct ncplane* n, const char* format, ...){
    va_list va;
    va_start(va, format);
    int ret = ncplane_vprintf(n, format, va);
    va_end(va);
    return ret;
}

static inline int ncplane_printf_yx(struct ncplane* n, int y, int x, const char* format, ...){
    va_list va;
    va_start(va, format);
    int ret = ncplane_vprintf_yx(n, y, x, format, va);
    va_end(va);
    return ret;
}

static inline int ncplane_printf_aligned(struct ncplane* n, int y, ncalign_e align, const char* format, ...){
    va_list va;
    va_start(va, format);
    int ret = ncplane_vprintf_aligned(n, y, align, format, va);
    va_end(va);
    return ret;
}

```

LISTING 61: Formatted output to planes.

Constant	Value	Comments
NCSTYLE_STANDOUT	0x00800000ul	Implementation-defined
NCSTYLE_UNDERLINE	0x00400000ul	
NCSTYLE_REVERSE	0x00200000ul	
NCSTYLE_BLINK	0x00100000ul	
NCSTYLE_DIM	0x00080000ul	Sometimes just a different color
NCSTYLE_BOLD	0x00040000ul	Sometimes just a different color
NCSTYLE_INVIS	0x00020000ul	
NCSTYLE_PROTECT	0x00010000ul	
NCSTYLE_ITALIC	0x01000000ul	Not commonly supported

TABLE 11: The NCSTYLE bits.

It is not guaranteed that all styles are usable on a given terminal emulator. As noted in Chapter 4.4, the `notcurses_supported_styles()` function can be used at runtime to determine which are available.

Bold mode is the culprit responsible for the widespread misconception that there are 16 or even 256 ANSI 12.5% of your framebuffer.

colors. Only 8 colors are defined in ECMA-48⁷⁵: black, red, green, yellow, blue, magenta, cyan, and white. Some terminals implemented bold and/or dim as additional colors, and these colors were often available for direct selection. A terminal can faithfully implement all of ANSI with only eight colors. `aixterm` supported 16 colors, which were picked up by `xterm` and `NCURSES` . `xterm` added support for 88- and 256-palette color in 1999⁷⁶.

9.3 The 64-bit channels value

In addition to the 32-bit `gcluster` and `attribute` fields, each `cell` expends a further 64 bits (half of its 16 byte total) on the `channels` field. Like `attribute` above, `channels` will be used as both an identifier and a type. Furthermore, a 64-bit `channels` variable is logically composed of two 32-bit `channel` values. The upper 32 bits describe the foreground channel, and the lower 32 bits describe the background channel. The full eight bytes are broken down in Listing 62.

```
// (channels & 0x8000000000000000ull): first column of wide EGC
// (channels & 0x4000000000000000ull): foreground is *not* "default color"
// (channels & 0x3000000000000000ull): foreground alpha (2 bits)
// (channels & 0x0800000000000000ull): foreground uses palette index
// (channels & 0x0700000000000000ull): reserved, must be 0
// (channels & 0x00ffffff00000000ull): foreground in 3x8 RGB (rrggbb)
// (channels & 0x0000000080000000ull): secondary column of wide EGC
// (channels & 0x0000000040000000ull): background is *not* "default color"
// (channels & 0x0000000030000000ull): background alpha (2 bits)
// (channels & 0x0000000080000000ull): background uses palette index
// (channels & 0x0000000070000000ull): reserved, must be 0
// (channels & 0x0000000000ffffffull): background in 3x8 RGB (rrggbb)
```

LISTING 62: Bits of the channels type

```
#define CELL_WIDEASIAN_MASK    0x8000000080000000ull
#define CELL_BGDEFAULT_MASK   0x4000000040000000ull
#define CELL_FGDEFAULT_MASK   (CELL_BGDEFAULT_MASK << 32u)
#define CELL_BG_MASK          0x000000000ffffffull
#define CELL_FG_MASK          (CELL_BG_MASK << 32u)
#define CELL_BG_PALETTE       0x0000000080000000ull
#define CELL_FG_PALETTE       (CELL_BG_PALETTE << 32u)
#define CELL_ALPHA_MASK       0x0000000030000000ull
#define CELL_ALPHA_SHIFT      28u
#define CELL_ALPHA_HIGHCONTRAST 3
#define CELL_ALPHA_TRANSPARENT 2
#define CELL_ALPHA_BLEND      1
#define CELL_ALPHA_OPAQUE     0
```

LISTING 63: Masks and other constants for working with channels

The first bit of either channel is high if and only if the cell is part of a multicolumn EGC. When a cell is loaded via e.g. `cell_load()`, the most significant bit (63) of `channels` is set if and only if the EGC is wide. No cell having a non-zero `gcluster` has the “secondary column of wide EGC” bit (31) set. This is set only in

⁷⁵All 8 colors are defined for both fore- and background.

⁷⁶Why 88? Ignoring possible connections to Neo-Nazis or *Back to the Future*...I’m not quite sure. The canonical 88-color palette is the 16 `aixterm` colors, a 4x4x4 color cube, and an 8-element grey ramp. Fair enough. But six bits can only represent 64 colors, and the seven necessary for 88 can represent 128. So why 88? Just to mirror the 16+6x6x6+16+24 of 256 colors? Perhaps the other space in a byte is used to encode additional information? I have been unable to reach a conclusive answer. Oh well. According to my calculations...when this baby hits 2²⁴, you’re gonna see some serious shit!

```

// Does the cell contain an East Asian Wide codepoint?
static inline bool cell_double_wide_p(const cell* c){
    return (c->channels & CELL_WIDEASIAN_MASK);
}

// Is this the right half of a wide character?
static inline bool cell_wide_right_p(const cell* c){
    return cell_double_wide_p(c) && c->gcluster == 0;
}

// Is this the left half of a wide character?
static inline bool cell_wide_left_p(const cell* c){
    return cell_double_wide_p(c) && c->gcluster;
}

```

LISTING 64: cell predicates for testing multicolumn properties.

framebuffer cells having a null glyph due to being “covered” by a preceding multicolumn EGC. These two bits should never be manipulated by the user; Notcurses automatically manages them, and relies upon them internally. You can of course read them, should you be so inclined. Three functions are supplied for testing these bits (Listing 64).

```

// Extract the 8-bit r/g/b components from a 32-bit channel.
static inline unsigned channel_r(unsigned channel){ return (channel & 0xff0000u) >> 16u; }
static inline unsigned channel_g(unsigned channel){ return (channel & 0x00ff00u) >> 8u; }
static inline unsigned channel_b(unsigned channel){ return (channel & 0x0000ffu); }

// Extract the three 8-bit R/G/B components from a 32-bit channel.
static inline unsigned channel_rgb(unsigned channel, unsigned* restrict r, unsigned* restrict g, unsigned* restrict b){
    *r = channel_r(channel);
    *g = channel_g(channel);
    *b = channel_b(channel);
    return channel;
}

// Set the three 8-bit components of a 32-bit channel, and mark it as not using the default color.
// Retain the other bits unchanged.
static inline int channel_set_rgb(unsigned* channel, int r, int g, int b){
    if(r >= 256 || g >= 256 || b >= 256){
        return -1;
    }
    if(r < 0 || g < 0 || b < 0){
        return -1;
    }
    unsigned c = (r << 16u) | (g << 8u) | b;
    *channel = (*channel & ^CELL_BG_MASK) | CELL_BGDEFAULT_MASK | c;
    return 0;
}

// Set the three 8-bit components of a 32-bit channel, and mark it as not using the default color. Retain the other bits
// unchanged. r, g, and b will be clipped to the range [0..255].
static inline void channel_set_rgb_clipped(unsigned* channel, int r, int g, int b){
    if(r >= 256){ r = 255; }
    if(g >= 256){ g = 255; }

```

```

    if(b >= 256){ b = 255; }
    if(r <= -1){ r = 0; }
    if(g <= -1){ g = 0; }
    if(b <= -1){ b = 0; }
    unsigned c = (r << 16u) | (g << 8u) | b;
    *channel = (*channel & ~CELL_BG_MASK) | CELL_BGDEFAULT_MASK | c;
}

// Same, but provide an assembled, packed 24 bits of rgb.
static inline int channel_set(unsigned* channel, unsigned rgb){
    if(rgb > 0xfffffu){
        return -1;
    }
    *channel = (*channel & ~CELL_BG_MASK) | CELL_BGDEFAULT_MASK | rgb;
    return 0;
}

// Extract the 2-bit alpha component from a 32-bit channel.
static inline unsigned channel_alpha(unsigned channel){
    return (channel & CELL_ALPHA_MASK) >> CELL_ALPHA_SHIFT;
}

// Set the 2-bit alpha component of the 32-bit channel.
static inline int channel_set_alpha(unsigned* channel, int alpha){
    if(alpha < CELL_ALPHA_OPAQUE || alpha > CELL_ALPHA_HIGHCONTRAST){
        return -1;
    }
    *channel = (alpha << CELL_ALPHA_SHIFT) | (*channel & ~CELL_ALPHA_MASK);
    if(alpha != CELL_ALPHA_OPAQUE){
        *channel |= CELL_BGDEFAULT_MASK;
    }
    return 0;
}

// Is this channel using the "default color" rather than RGB/palette-indexed?
static inline bool channel_default_p(unsigned channel){
    return !(channel & CELL_BGDEFAULT_MASK);
}

// Is this channel using palette-indexed color rather than RGB?
static inline bool channel_palindex_p(unsigned channel){
    return !channel_default_p(channel) && (channel & CELL_BG_PALETTE);
}

// Mark the channel as using its default color, which also marks it opaque.
static inline unsigned channel_set_default(unsigned* channel){
    return *channel &= ~(CELL_BGDEFAULT_MASK | CELL_ALPHA_HIGHCONTRAST);
}

```

LISTING 65: The full channel API.

The full channel API (suitable for dealing with a single 32-bit channel) is provided in Listing 65.

```

// Extract the 32-bit background channel from a channel pair.
static inline uint32_t channels_bchannel(uint64_t channels){
    return channels & 0xfffffffflu;
}

```



```

}

// Extract the 32-bit foreground channel from a channel pair.
static inline uint32_t channels_fchannel(uint64_t channels){
    return channels_bchannel(channels >> 32u);
}

// Set the 32-bit background channel of a channel pair.
static inline uint64_t channels_set_bchannel(uint64_t* channels, uint32_t channel){
    return *channels = (*channels & 0xffffffff000000llu) | channel;
}

// Set the 32-bit foreground channel of a channel pair.
static inline uint64_t channels_set_fchannel(uint64_t* channels, uint32_t channel){
    return *channels = (*channels & 0xffffffffllu) | ((uint64_t)channel << 32u);
}

static inline uint64_t channels_combine(uint32_t fchan, uint32_t bchan){
    uint64_t channels = 0;
    channels_set_fchannel(&channels, fchan);
    channels_set_bchannel(&channels, bchan);
    return channels;
}

// Extract 24 bits of foreground RGB from 'channels', shifted to LSBs.
static inline unsigned channels_fg(uint64_t channels){
    return channels_fchannel(channels) & CELL_BG_MASK;
}

// Extract 24 bits of background RGB from 'channels', shifted to LSBs.
static inline unsigned channels_bg(uint64_t channels){
    return channels_bchannel(channels) & CELL_BG_MASK;
}

// Extract 2 bits of foreground alpha from 'channels', shifted to LSBs.
static inline unsigned channels_fg_alpha(uint64_t channels){
    return channel_alpha(channels_fchannel(channels));
}

// Extract 2 bits of background alpha from 'channels', shifted to LSBs.
static inline unsigned channels_bg_alpha(uint64_t channels){
    return channel_alpha(channels_bchannel(channels));
}

// Extract 24 bits of foreground RGB from 'channels', split into subchannels.
static inline unsigned channels_fg_rgb(uint64_t channels, unsigned* r, unsigned* g, unsigned* b){
    return channel_rgb(channels_fchannel(channels), r, g, b);
}

// Extract 24 bits of background RGB from 'channels', split into subchannels.
static inline unsigned channels_bg_rgb(uint64_t channels, unsigned* r, unsigned* g, unsigned* b){
    return channel_rgb(channels_bchannel(channels), r, g, b);
}

```

```

// Set the r, g, and b channels for the foreground component of this 64-bit 'channels' variable, and mark
// it as not using the default color.
static inline int channels_set_fg_rgb(uint64_t* channels, int r, int g, int b){
    unsigned channel = channels_fchannel(*channels);
    if(channel_set_rgb(&channel, r, g, b) < 0){
        return -1;
    }
    *channels = ((uint64_t)channel << 32llu) | (*channels & 0xffffffffllu);
    return 0;
}

// Same, but clips to [0..255].
static inline void channels_set_fg_rgb_clipped(uint64_t* channels, int r, int g, int b){
    unsigned channel = channels_fchannel(*channels);
    channel_set_rgb_clipped(&channel, r, g, b);
    *channels = ((uint64_t)channel << 32llu) | (*channels & 0xffffffffllu);
}

// Set the r, g, and b channels for the background component of this 64-bit 'channels' variable, and mark
// it as not using the default color.
static inline int channels_set_bg_rgb(uint64_t* channels, int r, int g, int b){
    unsigned channel = channels_bchannel(*channels);
    if(channel_set_rgb(&channel, r, g, b) < 0){
        return -1;
    }
    channels_set_bchannel(channels, channel);
    return 0;
}

// Same, but clips to [0..255].
static inline void channels_set_bg_rgb_clipped(uint64_t* channels, int r, int g, int b){
    unsigned channel = channels_bchannel(*channels);
    channel_set_rgb_clipped(&channel, r, g, b);
    channels_set_bchannel(channels, channel);
}

// Same, but set an assembled 24 bit channel at once.
static inline int channels_set_fg(uint64_t* channels, unsigned rgb){
    unsigned channel = channels_fchannel(*channels);
    if(channel_set(&channel, rgb) < 0){
        return -1;
    }
    *channels = ((uint64_t)channel << 32llu) | (*channels & 0xffffffffllu);
    return 0;
}

static inline int channels_set_bg(uint64_t* channels, unsigned rgb){
    unsigned channel = channels_bchannel(*channels);
    if(channel_set(&channel, rgb) < 0){
        return -1;
    }
    channels_set_bchannel(channels, channel);
    return 0;
}

```

```

// Set the 2-bit alpha component of the foreground channel.
static inline int channels_set_fg_alpha(uint64_t* channels, int alpha){
    unsigned channel = channels_fchannel(*channels);
    if(channel_set_alpha(&channel, alpha) < 0){
        return -1;
    }
    *channels = ((uint64_t)channel << 32llu) | (*channels & 0xffffffffllu);
    return 0;
}

// Set the 2-bit alpha component of the background channel.
static inline int channels_set_bg_alpha(uint64_t* channels, int alpha){
    if(alpha == CELL_ALPHA_HIGHCONTRAST){ // forbidden for background alpha
        return -1;
    }
    unsigned channel = channels_bchannel(*channels);
    if(channel_set_alpha(&channel, alpha) < 0){
        return -1;
    }
    channels_set_bchannel(channels, channel);
    return 0;
}

// Is the foreground using the "default foreground color"?
static inline bool channels_fg_default_p(uint64_t channels){
    return channel_default_p(channels_fchannel(channels));
}

// Is the foreground using indexed palette color?
static inline bool channels_fg_palindex_p(uint64_t channels){
    return channel_palindex_p(channels_fchannel(channels));
}

// Is the background using the "default background color"? The "default background color"
// must generally be used to take advantage of terminal-effected transparency.
static inline bool channels_bg_default_p(uint64_t channels){
    return channel_default_p(channels_bchannel(channels));
}

// Is the background using indexed palette color?
static inline bool channels_bg_palindex_p(uint64_t channels){
    return channel_palindex_p(channels_bchannel(channels));
}

// Mark the foreground channel as using its default color.
static inline uint64_t channels_set_fg_default(uint64_t* channels){
    unsigned channel = channels_fchannel(*channels);
    channel_set_default(&channel);
    *channels = ((uint64_t)channel << 32llu) | (*channels & 0xffffffffllu);
    return *channels;
}

// Mark the foreground channel as using its default color.

```

```

static inline uint64_t channels_set_bg_default(uint64_t* channels){
    unsigned channel = channels_bchannel(*channels);
    channel_set_default(&channel);
    channels_set_bchannel(channels, channel);
    return *channels;
}

```

LISTING 66: The full channels API.

The full channels API (suitable for dealing with two single 32-bit channels in a single 64-bit) is provided in Listing 66. Remember, the foreground is always the more significant 32 bits of a 64-bit channel pair.

```

// Returns the result of blending two channels. 'blends' indicates how heavily 'c1' ought be weighed.
// If 'blends' is 0, 'c1' will be entirely replaced by 'c2'. If 'c1' is otherwise the default color,
// 'c1' will not be touched, since we can't blend default colors. Likewise, if 'c2' is a default
// color, it will not be used (unless 'blends' is 0).
//
// Palette-indexed colors do not blend, and since we need the attrword to store them, we just don't
// fuck wit' 'em here. Do not pass me palette-indexed channels! I will eat them.
static inline unsigned
channels_blend(unsigned c1, unsigned c2, unsigned* blends){
    if(channel_alpha(c2) == CELL_ALPHA_TRANSPARENT){
        return c1; // do *not* increment *blends
    }
    unsigned rsum, gsum, bsum;
    channel_rgb(c2, &rsum, &gsum, &bsum);
    bool c2default = channel_default_p(c2);
    if(*blends == 0){
        // don't just return c2, or you set wide status and all kinds of crap
        if(channel_default_p(c2)){
            channel_set_default(&c1);
        }else{
            channel_set_rgb(&c1, rsum, gsum, bsum);
        }
        channel_set_alpha(&c1, channel_alpha(c2));
    }else if(!c2default && !channel_default_p(c1)){
        rsum = (channel_r(c1) * *blends + rsum) / (*blends + 1);
        gsum = (channel_g(c1) * *blends + gsum) / (*blends + 1);
        bsum = (channel_b(c1) * *blends + bsum) / (*blends + 1);
        channel_set_rgb(&c1, rsum, gsum, bsum);
        channel_set_alpha(&c1, channel_alpha(c2));
    }
    ++*blends;
    return c1;
}

```

LISTING 67: Channel blending.

To round out our coverage of output functionality, `channels_blend()` is detailed in Listing 67. You won't typically call this function (though you can, to hand-blend colors), but it's a critical element in the rendering path.

10 Lines, boxes, and fills

10.1 Linear interpolation (“lerping”) and lines

On the plain behind him are the wanderers in search of bones and those who do not search and they move haltingly in the light like mechanisms whose movements are monitored with escapement and pallet so that they appear restrained by a prudence or reflectiveness which has no inner reality and they cross in their progress one by one that track of holes that runs to the rim of the visible ground and which seems less the pursuit of some continuance than the verification of a principle, a validation of sequence and causality as if each round and perfect hole owed its existence to the one before it there on that prairie upon which are the bones and the gatherers of bones and those who do not gather.

Cormac McCarthy, *Blood Meridian*

Following actual text, the most frequent need of a TUI is probably vertical and horizontal lines. Notcurses allows such lines to be drawn using arbitrary EGCs, though you’ll usually want one of the Unicode Box Drawing characters (see Figure 43) or the Block Elements (Figure 37).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2500	—	—			---	---					┌	┌	┌	┌
2510	┐	┐	┐	┐	└	└	└	└	┘	┘	┘	┘	└	└	└	└
2520	└	└	└	└	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘
2530	┘	┘	┘	┘	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐
2540	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘	┘
2550	=		┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌
2560	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌
2570	┌	/	\	X	-		-		-		-		-		-	

FIGURE 43: Unicode box-drawing characters (source: *Chininazu12*, public domain).

Using `ncplane_hline()` or `ncplane_vline()`, a single cell is provided, and this cell (including its attributes and channels) is reproduced on each cell of the line⁷⁷. Two additional forms are provided: `ncplane_hline_interp()` and `ncplane_vline_interp()` (see Listing 68). Both of these accept two channel pairs, and perform a linear

⁷⁷If you provide a multicolumn EGC...I have no idea what happens. Maybe it works? I should look into that, huh.

interpolation between the two foreground and two background channels. Providing the equivalent values for either channel will result in that channel remaining constant along the line’s length.

```
// Draw horizontal or vertical lines using the specified cell, starting at the current cursor position. The
// cursor will end at the cell following the last cell output (even, perhaps counter-intuitively, when
// drawing vertical lines), just as if ncplane_putc() was called at that spot. Return the number of cells
// drawn on success. On error, return the negative number of cells drawn.
int ncplane_hline_interp(struct ncplane* n, const cell* c, int len, uint64_t c1, uint64_t c2);

static inline int ncplane_hline(struct ncplane* n, const cell* c, int len){
    return ncplane_hline_interp(n, c, len, c->channels, c->channels);
}

int ncplane_vline_interp(struct ncplane* n, const cell* c, int len, uint64_t c1, uint64_t c2);

static inline int ncplane_vline(struct ncplane* n, const cell* c, int len){
    return ncplane_vline_interp(n, c, len, c->channels, c->channels);
}
```

LISTING 68: Functions for drawing lines.

10.2 Boxes

Rectangles are regularly required as borders and for grouping. Notcurses supports flexible box drawing. Boxes have their upper-left corner at the current cursor position, unless drawn with `ncplane_perimeter()`, which draws along the edges of the plane (and has its upper-left corner at the plane origin). Box-drawing functions accept six `cell` objects, one for each corner, one for horizontal lines, and one for vertical lines. It is possible to apply linear interpolation between the corners, in which case the colors of the horizontal and vertical line-drawing cells will be ignored. It is possible to draw none, all, or any set of the four corners and none, all, or any set of the four sides. These configurable behaviors are specified via the `ctlword` bitmask parameter. `ctlword` is defined in the least significant byte, where bits 4–7 are a gradient mask, and 0–3 are a border mask (see Table 12).

Constant	Bit	Property
<code>NCBOXMASK_TOP</code>	0x001	Inhibit top
<code>NCBOXMASK_RIGHT</code>	0x002	Inhibit right
<code>NCBOXMASK_BOTTOM</code>	0x004	Inhibit bottom
<code>NCBOXMASK_LEFT</code>	0x008	Inhibit left
<code>NCBOXGRAD_TOP</code>	0x010	Left side linear interpolation
<code>NCBOXGRAD_RIGHT</code>	0x020	Bottom side linear interpolation
<code>NCBOXGRAD_BOTTOM</code>	0x040	Right side linear interpolation
<code>NCBOXGRAD_LEFT</code>	0x080	Top side linear interpolation
<code>x</code>	0x100	Require 1 connecting edge to draw corner
<code>x</code>	0x200	Require 2 connecting edges to draw corner
<code>x</code>	0x300	Draw no corners

TABLE 12: `ctlword` parameter for box-drawing.

By default, vertexes are drawn whether their connecting edges are drawn or not. The value of the bits corresponding to `NCBOXCORNER_MASK` (0x300) control this, and are interpreted as the number of connecting edges necessary to draw a given corner. At 0 (the default), corners are always drawn. At 3, corners are never drawn (as at most 2 edges can touch a box’s corner).

It can be tedious to set up the six `cell` parameters to these functions. Since boxes are typically drawn with one of a small number of sets of EGCs, helper functions are provided for each set. I usually go with the

```

int ncplane_box(struct ncplane* n, const cell* ul, const cell* ur, const cell* ll, const cell* lr,
               const cell* hline, const cell* vline, int ystop, int xstop, unsigned ctlword);

// Draw a box with its upper-left corner at the current cursor position, having dimensions 'ylen'x'xlen'.
// See ncplane_box() for more information. The minimum box size is 2x2, and it cannot be drawn off-screen.
static inline int
ncplane_box_sized(struct ncplane* n, const cell* ul, const cell* ur, const cell* ll,
                 const cell* lr, const cell* hline, const cell* vline, int ylen, int xlen, unsigned ctlword){
    int y, x;
    ncplane_cursor_yx(n, &y, &x);
    return ncplane_box(n, ul, ur, ll, lr, hline, vline, y + ylen - 1, x + xlen - 1, ctlword);
}

static inline int
ncplane_perimeter(struct ncplane* n, const cell* ul, const cell* ur, const cell* ll,
                 const cell* lr, const cell* hline, const cell* vline, unsigned ctlword){
    if(ncplane_cursor_move_yx(n, 0, 0)){
        return -1;
    }
    int dimy, dimx;
    ncplane_dim_yx(n, &dimy, &dimx);
    return ncplane_box_sized(n, ul, ur, ll, lr, hline, vline, dimy, dimx, ctlword);
}

```

LISTING 69: Functions for drawing rectilinear boxes.

pleasantly rounded “Light Arc” Box Drawing codes (Listing 70). Or, should you prefer, there are the strong, sure Double Box Drawing characters (Listing 71).

10.3 Gradients and polyfills

`ncplane_polyfill()` should be applied to a coordinate with no glyph (Listing 72). That coordinate will be filled with the provided `cell`. The function then effectively recurses on all cardinaly connected coordinates, thus filling a bounded region with the provided cell. This operation is akin to the “flood fill” of pixel graphics. Sometimes you’ll want to destroy all content in the plane, reinitializing its framebuffer and base cell without changing the geometry. `ncplane_erase()` allows you to do this in one fell swoop, with the added bonus functionality of resetting the associated `egcpool`. A freshly-reset `egcpool` can be much faster than one which has been heavily used, requiring a search to find suitable free space. The state of the framebuffer is exactly as it was when the plane was created—all cells hold the null EGC, all attributes are 0, and all colors are defaults. All cells associated with this plane are invalidated, so be sure you’re not holding onto any. `ncplane_polyfill_yx()` and `ncplane_erase()` are detailed in Listing 72.

A single EGC and attribute can be written to a rectangular region in any of a single color, a vertical, horizontal, or diagonal gradient, or a 4-cornered “inverted radial” gradient (see Listing 73). The gradient operation is independently applied to both the fore- and background of 4 64-bit channel parameters. Palette-indexed color is not yet supported for gradients.

10.4 Blitting

Sometimes, you’ve got a chunk of RGBA or BGRx in memory, and just want to blast it onto a plane as quickly as possible. Blitting functions (see Listing 74) exist to transform such pixels into Unicode Block Elements. Every two input rows become a single row, using half-blocks when necessary. Columns are mapped 1-to-1. This functionality was used, for instance, to set up Notcurses as a rendering backend for NEStopia and RetroArch, and these functions form the heart of the multimedia functionality described in Chapter 11. Note that these functions do not offer any scaling capabilities.

```

static inline int cells_rounded_box(struct ncplane* n, uint32_t attr, uint64_t channels,
                                   cell* ul, cell* ur, cell* ll, cell* lr, cell* hl, cell* vl){
    return cells_load_box(n, attr, channels, ul, ur, ll, lr, hl, vl, "␣U-|");
}

static inline int ncplane_rounded_box(struct ncplane* n, uint32_t attr, uint64_t channels,
                                      int ystop, int xstop, unsigned ctlword){
    int ret = 0;
    cell ul = CELL_TRIVIAL_INITIALIZER, ur = CELL_TRIVIAL_INITIALIZER;
    cell ll = CELL_TRIVIAL_INITIALIZER, lr = CELL_TRIVIAL_INITIALIZER;
    cell hl = CELL_TRIVIAL_INITIALIZER, vl = CELL_TRIVIAL_INITIALIZER;
    if((ret = cells_rounded_box(n, attr, channels, &ul, &ur, &ll, &lr, &hl, &vl)) == 0){
        ret = ncplane_box(n, &ul, &ur, &ll, &lr, &hl, &vl, ystop, xstop, ctlword);
    }
    cell_release(n, &ul); cell_release(n, &ur);
    cell_release(n, &ll); cell_release(n, &lr);
    cell_release(n, &hl); cell_release(n, &vl);
    return ret;
}

static inline int
ncplane_rounded_box_sized(struct ncplane* n, uint32_t attr, uint64_t channels, int ylen, int xlen, unsigned ctlword){
    int y, x;
    ncplane_cursor_yx(n, &y, &x);
    return ncplane_rounded_box(n, attr, channels, y + ylen - 1, x + xlen - 1, ctlword);
}

```

LISTING 70: Helpers for rounded-corner boxes.

10.5 Staining

Consult Chapter 9.1 for the “stainable” family of text output functions, allowing EGCs to be replaced without affecting attributes or channels. To modify attributes or channels in isolation but *en masse*, Notcurses provides `ncplane_format()` and `ncplane_stain()`.

Staining could be accomplished without a special function call by reflecting on each cell of interest using `ncplane_at_yx()`, changing the cell, and writing it back, but `ncplane_stain()` and `ncplane_format()` are faster and simpler. Creating a new plane of all null glyphs, setting the channels as desired, and placing it atop the region would also accomplish a staining, and can further be used as a “moving” stain. This wouldn’t affect the lower plane; it would purely be an effect of rendering. It is not possible to simulate `ncplane_format()` in this way, however—glyph and attribute are a single dimension and cannot be independently transparent.


```

static inline int cells_double_box(struct ncplane* n, uint32_t attr, uint64_t channels,
                                  cell* ul, cell* ur, cell* ll, cell* lr, cell* hl, cell* vl){
    return cells_load_box(n, attr, channels, ul, ur, ll, lr, hl, vl, "▣=|");
}

static inline int ncplane_double_box(struct ncplane* n, uint32_t attr, uint64_t channels,
                                     int ystop, int xstop, unsigned ctlword){
    int ret = 0;
    cell ul = CELL_TRIVIAL_INITIALIZER, ur = CELL_TRIVIAL_INITIALIZER;
    cell ll = CELL_TRIVIAL_INITIALIZER, lr = CELL_TRIVIAL_INITIALIZER;
    cell hl = CELL_TRIVIAL_INITIALIZER, vl = CELL_TRIVIAL_INITIALIZER;
    if((ret = cells_double_box(n, attr, channels, &ul, &ur, &ll, &lr, &hl, &vl)) == 0){
        ret = ncplane_box(n, &ul, &ur, &ll, &lr, &hl, &vl, ystop, xstop, ctlword);
    }
    cell_release(n, &ul); cell_release(n, &ur);
    cell_release(n, &ll); cell_release(n, &lr);
    cell_release(n, &hl); cell_release(n, &vl);
    return ret;
}

static inline int ncplane_double_box_sized(struct ncplane* n, uint32_t attr, uint64_t channels,
                                           int ylen, int xlen, unsigned ctlword){
    int y, x;
    ncplane_cursor_yx(n, &y, &x);
    return ncplane_double_box(n, attr, channels, y + ylen - 1, x + xlen - 1, ctlword);
}

```

LISTING 71: Helpers for doubly-thick boxes.

```

// Starting at the specified coordinate, if it has no glyph, 'c' is copied into it. We do the same to
// all cardinaly-connected glyphless cells, filling in everything behind a boundary. Returns the
// number of cells polyfilled. An invalid initial y, x is an error.
int ncplane_polyfill_yx(struct ncplane* n, int y, int x, const cell* c);

void ncplane_erase(struct ncplane* n);

```

LISTING 72: Polyfills and plane erasure.

```

// Draw a gradient with its upper-left corner at the current cursor position, stopping at 'ystop'x'xstop'.
// The glyph composed of 'egc' and 'attrword' is used for all cells. The channels specified by 'ul', 'ur',
// 'll', and 'lr' are composed into foreground and background gradients. To do a vertical gradient, 'ul'
// ought equal 'ur' and 'll' ought equal 'lr'. To do a horizontal gradient, 'ul' ought equal 'll' and 'ur'
// ought equal 'ul'. To color everything the same, all four channels should be equivalent. The resulting
// alpha values are equal to incoming alpha values.
//
// Preconditions for gradient operations (error otherwise):
//
// all: only RGB colors, unless all four channels match as default
// all: all alpha values must be the same
// 1x1: all four colors must be the same
// 1xN: both top and both bottom colors must be the same (vertical gradient)
// Nx1: both left and both right colors must be the same (horizontal gradient)
int ncplane_gradient(struct ncplane* n, const char* egc, uint32_t attrword, uint64_t ul, uint64_t ur,
                    uint64_t ll, uint64_t lr, int ystop, int xstop);

// Do a high-resolution gradient using upper blocks and synced backgrounds. This doubles the number of
// vertical gradations, but restricts you to half blocks (appearing to be full blocks).
int ncplane_highgradient(struct ncplane* n, uint32_t ul, uint32_t ur,
                        uint32_t ll, uint32_t lr, int ystop, int xstop);

// Draw a gradient with its upper-left corner at the current cursor position, having dimensions
// 'ylen'x'xlen'. See ncplane_gradient for more information.
static inline int ncplane_gradient_sized(struct ncplane* n, const char* egc, uint32_t attrword, uint64_t ul,
                                        uint64_t ur, uint64_t ll, uint64_t lr, int ylen, int xlen){

    if(ylen < 1 || xlen < 1){
        return -1;
    }
    int y, x;
    ncplane_cursor_yx(n, &y, &x);
    return ncplane_gradient(n, egc, attrword, ul, ur, ll, lr, y + ylen - 1, x + xlen - 1);
}

static inline int ncplane_highgradient_sized(struct ncplane* n, uint64_t ul, uint64_t ur,
                                            uint64_t ll, uint64_t lr, int ylen, int xlen){

    if(ylen < 1 || xlen < 1){
        return -1;
    }
    int y, x;
    ncplane_cursor_yx(n, &y, &x);
    return ncplane_highgradient(n, ul, ur, ll, lr, y + ylen - 1, x + xlen - 1);
}

```

LISTING 73: Drawing gradients.

```
// Blit a flat array 'data' of BGRx 32-bit values to the nplane 'nc', offset from the upper left by 'placey' and 'placex'.
// Each row ought occupy 'linesize' bytes (this might be greater than lenx * 4 due to padding). A subregion of the input
// can be specified with 'begy'x'begx' and 'leny'x'lenx'.
int ncblit_bgrx(struct nplane* nc, int placey, int placex, int linesize, const unsigned char* data,
               int begy, int begx, int leny, int lenx);

// Blit a flat array 'data' of RGBA 32-bit values to the nplane 'nc', offset from the upper left by 'placey' and 'placex'.
// Each row ought occupy 'linesize' bytes (this might be greater than lenx * 4 due to padding). A subregion of the input can
// be specified with 'begy'x'begx' and 'leny'x'lenx'.
int ncblit_rgba(struct nplane* nc, int placey, int placex, int linesize, const unsigned char* data,
                int begy, int begx, int leny, int lenx);
```

LISTING 74: Blitting BGRx and RGBA.

```
// Set the given style throughout the specified region, keepying content and channels otherwise unchanged.
int ncplane_format(struct nplane* n, int ystop, int xstop, uint32_t attrword);

// Set the given channels throughout the specified region, keepying content and attributes otherwise unchanged.
int ncplane_stain(struct nplane* n, int ystop, int xstop, uint64_t ul, uint64_t ur, uint64_t ll, uint64_t lr);
```

LISTING 75: Changing attributes or channels in isolation.

11 Multimedia (images and videos)

Media decoding and scaling is handled by libAV from FFmpeg, resulting in a `ncvisual` object. This object generates frames, each one corresponding to a renderable scene on the associated plane. If Notcurses is built without FFmpeg support, these functions will all return error. The flow of multimedia from the view of a Notcurses application is:

- The media is opened with `ncplane_visual_open()` or `ncvisual_open_plane`. The latter creates a new plane suitable for rendering the media; the former will draw to a preexisting plane. Stream format and the codecs in use are determined during this step, but not necessarily the geometry of the visual data.
- A frame is decoded. Several frames might have been decoded from the underlying stream, but the application sees them one at a time. This frame carries geometry information along with a flat matrix of pixels, along with some manner of timing information⁷⁸. **begin loop**
- (Optional) Any subtitles associated with the frame are retrieved.
- The frame may be rendered. A delay might be taken. **end loop**

```
// Open a visual (image or video), associating it with the specified ncplane.
// Returns NULL on any error, writing the AVError to 'averr'.
struct ncvisual* ncplane_visual_open(struct ncplane* nc, const char* file, int* averr);

// Destroy an ncvisual. Rendered elements will not be disrupted, but the visual
// can be neither decoded nor rendered any further.
void ncvisual_destroy(struct ncvisual* ncv);

// Return the plane to which this ncvisual is bound.
struct ncplane* ncvisual_plane(struct ncvisual* ncv);
```

LISTING 76: Opening and destroying multimedia with `ncvisual`.

Any scaling is applied during the decoding step. It is thus sadly not possible to redraw decoded media at a different size. This isn't usually a problem in practice, since streaming media will provide a new frame (at the correct size) shortly, and single-frame media can simply be decoded afresh.

```
// extract the next frame from an ncvisual. returns NULL on end of file,
// writing AVERROR_EOF to 'averr'. returns NULL on a decoding or allocation
// error, placing the AVError in 'averr'. this frame is invalidated by a
// subsequent call to ncvisual_decode(), and should not be freed by the caller.
struct AVFrame* ncvisual_decode(struct ncvisual* nc, int* averr);

// Render the decoded frame to the associated ncplane. The frame will be scaled
// to the size of the ncplane per the ncscale_e style. A subregion of the
// frame can be specified using 'begx', 'begy', 'lenx', and 'leny'. To render
// the rectangle formed by begy x begx and the lower-right corner, zero can be
// supplied to 'leny' and 'lenx'. Zero for all four values will thus render the
// entire visual. Negative values for any of the four parameters are an error.
// It is an error to specify any region beyond the boundaries of the frame.
int ncvisual_render(const struct ncvisual* ncv, int begy, int begx, int leny, int lenx);
```

LISTING 77: Decoding and rendering multimedia with `ncvisual`.

Subtitles (considered “metadata” in FFmpeg) aren't advertised in any way to the caller. If subtitles are to be displayed, you can simply call `ncvisual_subtitle()` at each frame. If a non-NULL string is returned, it is valid

⁷⁸This timing information can come in three different forms: “frame number” (which can be multiplied by some known frames per second to get a time offset), “offset” in some unit of time, and “time to display” for this frame in some unit of time. This last sadly cannot yield an O(1) time offset within the stream.

UTF-8 subtitle text. The subtitle will *not* typically be repeated for all frames where it ought be displayed, so it's best left persistent. Unfortunately, there's no good way to know when it ought be struck from the display. Alas!

```
// If a subtitle ought be displayed at this time, returns a heap-allocated copy
// of the UTF8 text. Otherwise returns NULL.
char* ncvisual_subtitle(const struct ncvisual* ncv);
```

LISTING 78: Acquiring subtitles.

11.1 Streaming video/animated GIFs.

```
// Called for each frame rendered from 'ncv'. If anything but 0 is returned, the streaming operation
// ceases immediately, and that value is propagated out.
typedef int (*streamcb)(struct notcurses* nc, struct ncvisual* ncv, void*);

// Shut up and display my frames! Provide as an argument to ncvisual_stream(). If you'd like subtitles to
// be decoded, provide an ncplane as the curry. If the curry is NULL, subtitles will not be displayed.
static inline int ncvisual_simple_streamer(struct notcurses* nc, struct ncvisual* ncv, void* curry){
    if(notcurses_render(nc)){
        return -1;
    }
    int ret = 0;
    if(curry){
        // need a cast for C++ callers
        struct ncplane* subncp = (struct ncplane*)curry;
        char* subtitle = ncvisual_subtitle(ncv);
        if(subtitle){
            if(ncplane_putstr_yx(subncp, 0, 0, subtitle) < 0){
                ret = -1;
            }
            free(subtitle);
        }
    }
    return ret;
}
```

LISTING 79: streamcb callback type and ncvisual_simple_streamer().

Running the loop discussed above is good for the soul, but usually it's sufficient to let Notcurses handle things, perhaps with a callback to your program. The function `ncvisual_stream()` (Listing 80) does just that, honoring the timing hints embedded in the stream as best it can. Use of this function is strongly recommended. For the simplest use, `ncvisual_simple_streamer()` (Listing 79) can be provided as the callback. It will print subtitles in the upper left, and otherwise render frames as it receives them. If you require more complex per-frame activity, provide your own callback of type `streamcb`, which can carry a `void*` curry.

11.2 Scaling images and video

Notcurses relies on FFmpeg for all scaling, and thus must tell it the size of the rendering area. An `ncvisual` is always scaled according to the geometry of its associated plane's entirety. The scaling target is twice the rendering target's height in rows, and equal to the rendering target's width in columns. If the media is smaller than this, it will be scaled up. If larger, it will be scaled down. A lossless representation thus requires rendering to a plane having exactly half the media's height, and its exact width. Since it's not generally possible to know the media's size until it's been partially decoded, Notcurses provides `ncvisual_open_plane()`, which creates a new plane based on the media's dimensions (this plane can be retrieved from the returned

```
// Stream the entirety of the media, according to its own timing. Blocking, obviously. streamer may be NULL;
// it is otherwise called for each frame, and its return value handled as outlined for stream cb. If
// streamer() returns non-zero, the stream is aborted, and that value is returned. By convention, return a
// positive number to indicate intentional abort from within streamer(). 'timescale' allows the frame
// duration time to be scaled. For a visual naturally running at 30FPS, a 'timescale' of 0.1 will result in
// 300FPS, and a 'timescale' of 10 will result in 3FPS. It is an error to supply 'timescale' less than or
// equal to 0.
int ncvisual_stream(struct notcurses* nc, struct ncvisual* ncv, int* avert,
                   float timescale, streamcb streamer, void* curry);
```

LISTING 80: Media streaming and `ncvisual_simple_streamer()`.

```
// How to scale the visual in ncvisual_from_file(). NCSCALE_NONE will open a plane tailored to the visual's
// exact needs, which is probably larger than the visible screen (but might be smaller). NCSCALE_SCALE
// scales a visual larger than the visible screen down, maintaining aspect ratio. NCSCALE_STRETCH stretches
// and scales the image in an attempt to fill the visible screen.
typedef enum { NCSCALE_NONE, NCSCALE_SCALE, NCSCALE_STRETCH, } ncscale_e;

// Open a visual, extract a codec and parameters, and create a new plane suitable for its display at 'y','x'.
// If there is sufficient room to display the visual in its native size, or if NCSCALE_NONE is passed for
// 'style', the new plane will be exactly that large. Otherwise, the plane will be as large as possible (given
// the visible screen), either maintaining aspect ratio (NCSCALE_SCALE) or abandoning it (NCSCALE_STRETCH).
struct ncvisual* ncvisual_from_file(struct notcurses* nc, const char* file, int* avert, int y, int x, ncscale_e style);
```

LISTING 81: Scaling media onto a new plane.

`ncvisual` using `ncvisual_plane()`. For an exactly-matched plane, supply `NCSCALE_NONE` (this plane might of course be bigger or smaller than the viewing area). Use `NCSCALE_STRETCH` to scale the image to the rendering area (just like `ncplane_visual_open()`, except with a new plane). `NCSCALE_SCALE` is not yet implemented, but will retain media aspect ratio whilst scaling to fit at least one dimension of the rendering area.

If the plane on which your `ncvisual` is to be rendered is larger than the rendering area, you can save time by rendering only a portion of the decoded image. `ncvisual_render()` accepts four arguments specifying a rectangular subsection via origin and dimensions. The last two arguments can be specified as `-1` to render through the end of the decoded frame. This is used in the `eagle` demo to progressively “zoom” in on a level map much larger than a typical terminal; in general, this will be the most efficient means of effecting parallax scrolling.

11.3 Sprites

Danny Hillis is probably best known to computer architects for Thinking Machines, but to the great video game-playing masses, it was his coinage of the term “sprite” which will be remembered (or not)[88]. First used in conjunction with the hallowed TMS34010 of Texas Instruments, sprites are simply bitmaps composed into a larger scene; like the French *esprit* or Celtic *spriggan*⁷⁹, they float ethereally above the ground (plane). Notcurses supports full-sized sprites via the combination of transparency and independently moved planes, often populated via an external image file (texture).

The basic recipe for a sprite is:

- Create a new plane above whatever background is in use.
- Set the base character of this plane transparent.
- Draw your object, possibly via `ncvisual_render()`.

⁷⁹Both derived from the Latin *spiritus*.

The new plane can be moved freely, and the base transparency will allow the underlying background to show through wherever you haven't drawn. The plane can be as large as is convenient, so long as it's transparent everywhere the sprite isn't present. To test whether the sprite has been "hit", the plane itself can be used as a bounding box, but much better accuracy can be had by testing the plane for character presence with `ncplane_at_yx()`.

If your sprite has an animation cycle—common for e.g. walking figures—it can often be easiest to render each frame of the cycle to a distinct plane, and keep only one visible at a time. This is used in the `luigi` demo for Luigi's running cycle.

12 Collecting and dispatching input

The X server primarily handles two kinds of hardware: input devices and output devices. Surprisingly, the input handling tends to be the more difficult and complicated of the two. Input is multi-source, concurrent, and highly dependent on complex user preferences.

The New X Developer's Guide[27]

To enter arbitrary Unicode using the keyboard, try pressing Ctrl+Shift+u. If successful, you ought see an underlined or otherwise stylized lowercase 'u'. The Unicode code point can be entered (each number will show up as you type it), followed by Enter. The sequence will then reduce to an EGC.

The most fundamental call is `notcurses_getc()`. This can operate as a non-blocking, timed, or blocking call. Provide a `NULL` struct `timespec 'ts'`. `notcurses_getc()` to block until input is received, or the call is interrupted by a signal (prepare the `sigset_t` parameter to mask signals as necessary). Provide the desired timeout in `'ts'` for a timed call, or zero out `'ts'` for a pure nonblocking call. On timeout, 0 is returned. On an error, -1 is returned. Otherwise, a `char32_t` is returned carrying a single UTF-32 Unicode codepoint. If the `ncinput` parameter `'ni'` is not `NULL`, it will be filled in with the codepoint, any applicable keyboard modifiers, and the cell of the input⁸⁰. Two helpers exist to simplify standard use cases: `notcurses_getc_nblock()` and `notcurses_getc_blocking()` do exactly what you'd expect.

It's important to note the difference in granularity of display and input. For display, we must write a UTF-8 encoded EGC as a single unit. Repeated calls do not combine within a cell. On input, however, we use UTF-32 and read a single code point, *not* an entire EGC. This is done to simplify input processing. It is recommended that, should you find yourself needing to implement backspace functionality, you implement it at the EGC level.

Notcurses places the terminal into `cbreak` mode (recall Chapter 6). This means that input is made immediately available, without waiting for a newline. The translation of certain key combinations (as specified by `termios`) into signals by the kernel line discipline still occurs...for now. It is likely that Notcurses will move to raw mode by the 2.0.0 release, in which case it will have to handle signal translation itself⁸¹.

Mouse events are reported only after a successful call to `notcurses_mouse_enable()`, and will no longer be reported following `notcurses_mouse_disable()`. Even when enabled, events are only returned while a button is held. There will be one event for the initial button press, one event for each cell into which the mouse moves while holding down the button, and one when the button is released. See Listing 83.

Input functions may be called concurrently with any output or read-only functions, but only one thread at a time may call into the input layer via any of its entry points.

There are many keys on a typical PC-104 keyboard which do not correspond to any Unicode code point. Examples include function keys, arrow keys, all the mouse buttons, and PageUp/PageDown⁸². For this purpose, Notcurses takes a chunk of Unicode's Private Supplementary Area-B, a plane explicitly left empty for use with local definitions. The lengthy set of `NCKEY_` macros are exported to allow trivial matching against these "synthesized characters". Finally, `NCKEY_RESIZE` is generated and pushed into the input queue whenever `SIGWINCH` is handled.

Input is not bound to any particular plane, and is indeed processed only minimally by Notcurses. The input queue is associated with a `notcurses` context as a whole. In particular, UI widgets do not automatically handle "their" input. Creating an `ncmenu` will see the menu drawn, but clicking it won't take any action by itself. Instead, each widget type provides a `*_handle_input()` function. Typically, your program should get the

⁸⁰Coordinates are currently reported only for pointing devices, not keyboards.

⁸¹The purpose of this planned change is to reliably acquire the state of keyboard modifiers—Shift, Ctrl, Alt, and their ilk. Currently, Notcurses reliably reports these modifiers only for mouse events. Furthermore, it is not currently possible to differentiate e.g. Ctrl+I from TAB (recall Table 5), but it will be once Notcurses begins using raw mode.

⁸²Though there *are* code points for Page Up (†) and Down(‡), I've never seen them emitted by the keyboard driver.


```

// See ppoll(2) for more detail. Provide a NULL 'ts' to block at length, a 'ts' of 0 for non-blocking operation,
// and otherwise a timespec to bound blocking. Signals in sigmask (less several we handle internally) will be
// atomically masked and unmasked per ppoll(2). It should generally contain all signals. Returns a single Unicode
// code point, or (char32_t)-1 on error. 'sigmask' may be NULL. Returns 0 on a timeout. If an event is processed,
// the return value is the 'id' field from that event. 'ni' may be NULL.
char32_t notcurses_getc(struct notcurses* n, const struct timespec* ts, sigset_t* sigmask, ncinput* ni);

// 'ni' may be NULL if the caller is uninterested in event details. If no event is ready, returns 0.
static inline char32_t notcurses_getc_nblock(struct notcurses* n, ncinput* ni){
    sigset_t sigmask;
    sigfillset(&sigmask);
    struct timespec ts = { .tv_sec = 0, .tv_nsec = 0 };
    return notcurses_getc(n, &ts, &sigmask, ni);
}

// 'ni' may be NULL if the caller is uninterested in event details.
// Blocks until an event is processed or a signal is received.
static inline char32_t notcurses_getc_blocking(struct notcurses* n, ncinput* ni){
    sigset_t sigmask;
    sigemptyset(&sigmask);
    return notcurses_getc(n, NULL, &sigmask, ni);
}

```

LISTING 82: Input can be acquired in nonblocking, blocking, or timed fashion.

```

// Enable the mouse in "button-event tracking" mode with focus detection and UTF8-style extended coordinates. On
// failure, -1 is returned. On success, 0 is returned, and mouse events will be published to notcurses_getc().
int notcurses_mouse_enable(struct notcurses* n);

// Disable mouse events. Any events in the input queue can still be delivered.
int notcurses_mouse_disable(struct notcurses* n);

```

LISTING 83: Mouse events must be explicitly enabled, and can be disabled.

input, check it against any input handling your program performs, and invoke `*handle_input()` appropriately for each UI widget active. Perform these checks in whatever order is appropriate for your program’s concept of “focus”. If the widget’s input handler returns true, consider the input “consumed” by that widget.

Just because a widget *consumes* input doesn’t mean it *acts upon* that input. Essentially, if the input results in actions limited to the widget, those actions are performed; if the input would result in control flow external to the widget, they are not. I know, I know. For now (as of Notcurses 1.2.3):

- Selector and multiselect widgets effect scrolling (at both the line and page level) themselves, using keyboard or mouse. Multiselect effects toggling itself, using keyboard or mouse. You must destroy the widget, and get its current state before doing so.
- Menu effects all movement and shading/unshading, using keyboard or mouse. You must get its current state when an item is executed.
- There does not yet exist a `reel_handle_input()` function. You must manipulate reels directly.

If you think this an unsatisfactory state of affairs, you’re not alone. Expect changes to the input system no later than Notcurses 2.0.0.

13 UI widgets

Widgets provide ready-made tools for acquiring user input or displaying data. As of Notcurses 1.2.4, there are four widgets: selector, multiselector, menu, and reel. Multiple widgets can be in use at one time, and they are drawn onto planes like any other output. Widgets can thus be moved up and down the z-axis. It is not recommended to scribble on widgets, but nothing prevents you from doing so. See Chapter 12 for information about routing input to widgets.

13.1 Selectors and multiselectors

The selector widget is an ncplane with a body section and optional title riser. The body section is populated with options and descriptions, and supports infinite scrolling up and down. The widget is automatically sized according to the largest input provided. The keyboard and mouse wheel can scroll through selections, and clicking on the arrows also scrolls. Selection and cancellation are implemented by the caller. The currently-selected option can be retrieved at any time. Option/description pairs can be added or removed while the widget is active, even if the removed pair is currently selected. Removing the last pair does not destroy the widget, and it is possible to create the widget with no pairs.

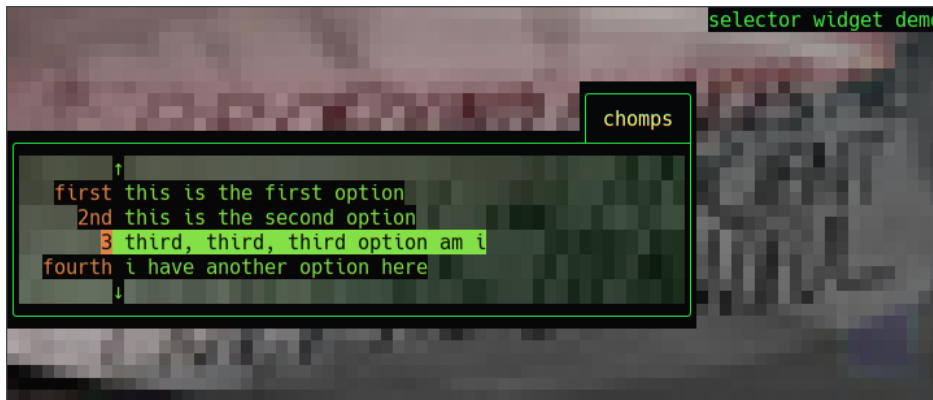


FIGURE 44: Naked selector.

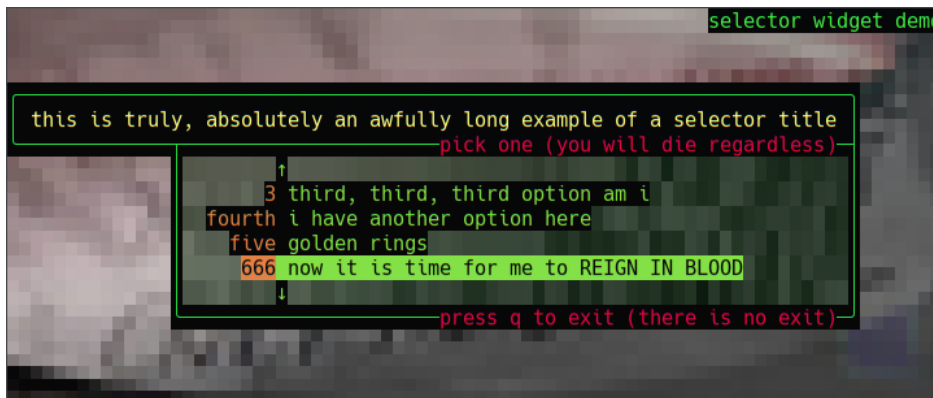


FIGURE 45: Selector with a long title.

13.2 Menus

Horizontal menu bars are supported on the top and bottom rows of planes. If a menu bar is longer than the bound plane, it will be only partially visible, but any unrolled section will be visible. Menus may be either visible or invisible by default. Set the 'hiding' option to get an invisible menu. In the event of a screen resize, menus will be automatically moved/resized.

```

struct selector_item {
    char* option;
    char* desc;
};

typedef struct selector_options {
    char* title; // title may be NULL, inhibiting riser, saving two rows.
    char* secondary; // secondary may be NULL
    char* footer; // footer may be NULL
    struct selector_item* items; // initial items and descriptions
    unsigned itemcount; // number of initial items and descriptions
    // default item (selected at start), must be < itemcount unless 'itemcount'
    // is 0, in which case 'defidx' must also be 0
    unsigned defidx;
    // maximum number of options to display at once, 0 to use all available space
    unsigned maxdisplay;
    // exhaustive styling options
    uint64_t opchannels; // option channels
    uint64_t descchannels; // description channels
    uint64_t titlechannels; // title channels
    uint64_t footchannels; // secondary and footer channels
    uint64_t boxchannels; // border channels
    uint64_t bgchannels; // background channels, used only in body
} selector_options;

struct ncselector* ncselector_create(struct ncplane* n, int y, int x, const selector_options* opts);

```

LISTING 84: Selector creation.

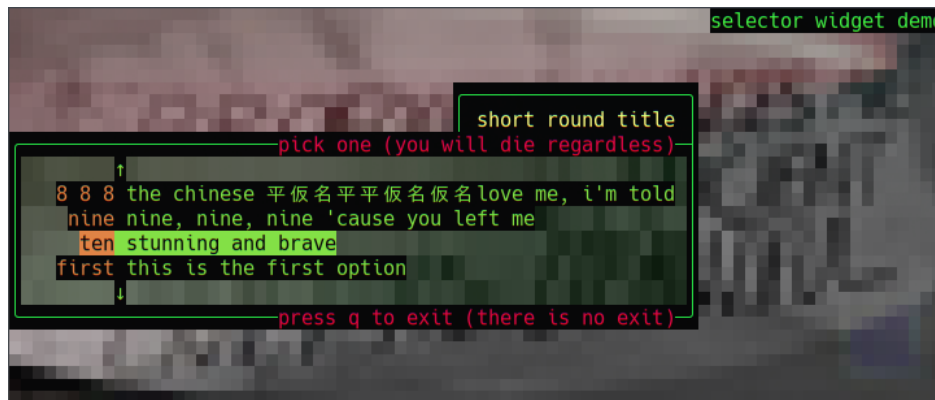


FIGURE 46: Short title intersecting with header.

Listing 86 covers creation of menus, and Listing 87 covers their control.

13.3 Reels

The `ncree`⁸³ is a UI abstraction supported by Notcurses in which dynamically-created and -destroyed toplevel entities (referred to as tablets) are arranged on a “cylinder”, allowing for infinite scrolling (infinite scrolling can be disabled, resulting in a rectangle rather than a cylinder). This works naturally with keyboard navigation, mouse scrolling wheels, and touchpads (including the capacitive touchscreens of modern cell

⁸³The term “reel” is borrowed from slot machines.

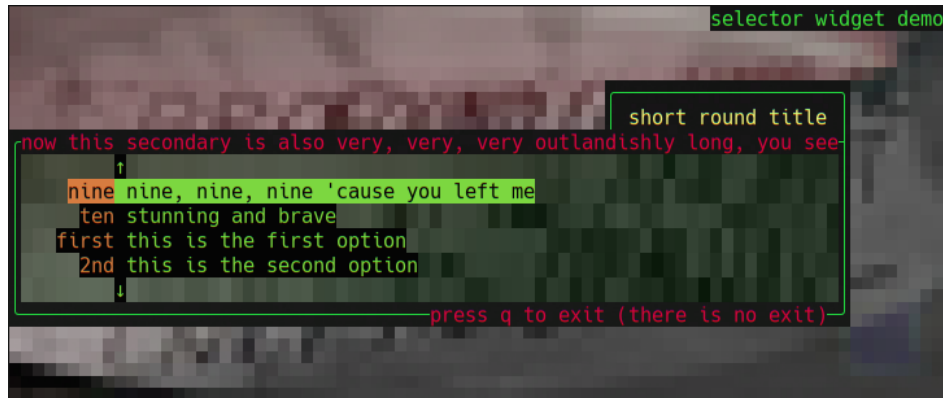


FIGURE 47: Selector with a long header.

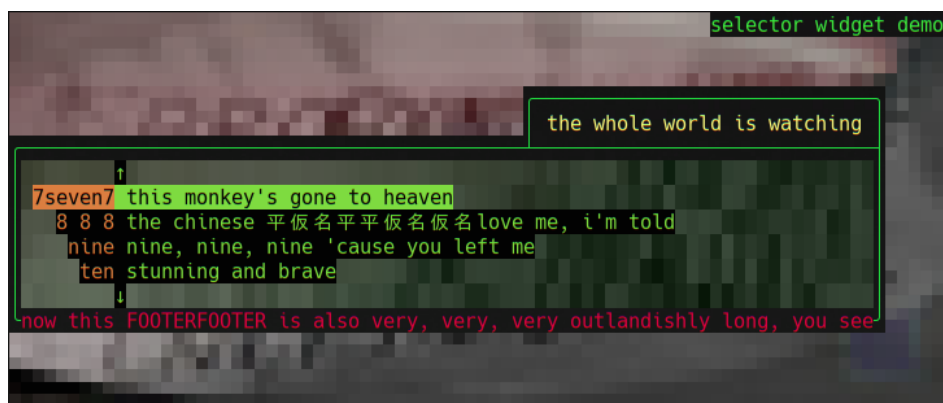


FIGURE 48: Selector with a long footer and no header.

phones). An ncreel initially has no tablets; at any given time thereafter, it has zero or more tablets, and if there is at least one tablet, one tablet is focused (and on-screen). If the last tablet is removed, no tablet is focused. A tablet can support navigation within the tablet, in which case there is an in-tablet focus for the focused tablet, which can also move among elements within the tablet. Reels have a `struct ncreel_options` object, passed to `ncreel_create()`; this struct is detailed in Listing 88.

The ncreel object tracks the size, number, information depth, and order of tablets, and the foci. It also draws the optional borders around tablets and the optional border of the reel itself. It knows nothing about the actual content of a tablet, save the number of lines it occupies at each information depth. The typical control flow is that an application receives events (from the UI or other event sources), and calls `ncreel_touch()` on tablets needing updates⁸⁴. Eventually, the application calls `ncreel_redraw()` to update the reel in its entirety⁸⁵. Notcurses will call into the application for some number of tablets, asking it to draw some line(s) from some tablet(s) at some particular coordinate of that tablet's panel. Finally, control returns to the application, and the cycle starts anew. The typedef for these callbacks is defined in Listing 89.

Each tablet might be wholly, partially, or not on-screen. Notcurses always places as much of the focused tablet as is possible on-screen (if the focused tablet has more lines than the actual reel does, it cannot be wholly on-screen. In this case, the focused subelements of the tablet are always on-screen). The placement of the focused tablet depends on how it was reached (when moving to the next tablet, offscreen tablets are brought onscreen at the bottom. When moving to the previous tablet, offscreen tablets are brought onscreen at the top. When moving to an arbitrary tablet which is neither the next nor previous tablet, it will be

⁸⁴This is one of the rare functions which can be called concurrently with `notcurses_render()`.

⁸⁵A call to `notcurses_render()` is still required to update the display.

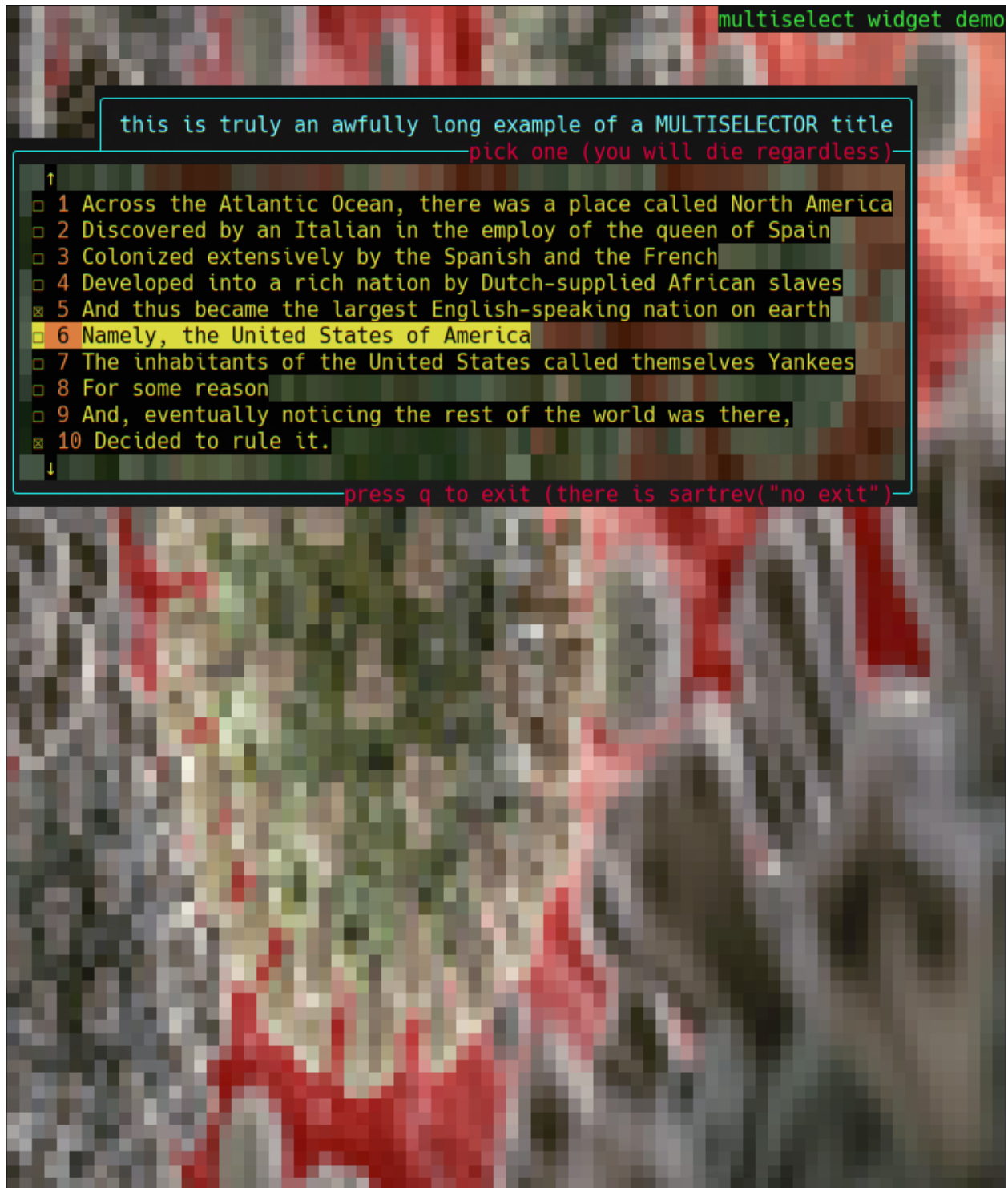


FIGURE 49: Multiselect.

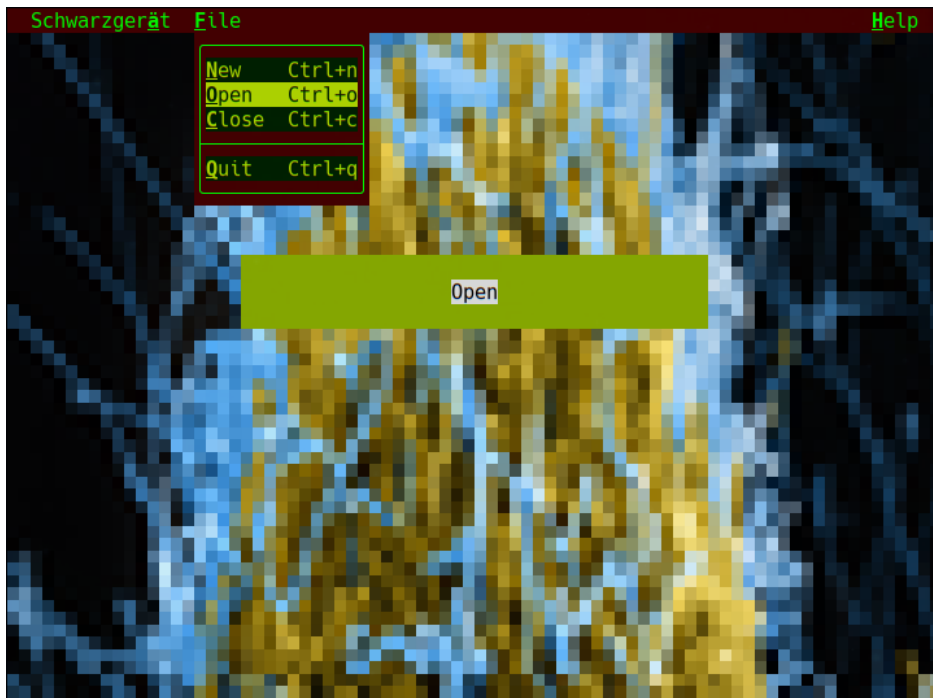


FIGURE 50: Menu along the top of the standard plane.



FIGURE 51: Menu along the bottom of the standard plane.



FIGURE 52: The notcurses-demo menu, unrolled *in media res*. Luigi, pursued by WarMECH, is leaping through the “Help” menu. In the upper left is the HUD, and at the bottom the About text, both implemented as translucent planes.

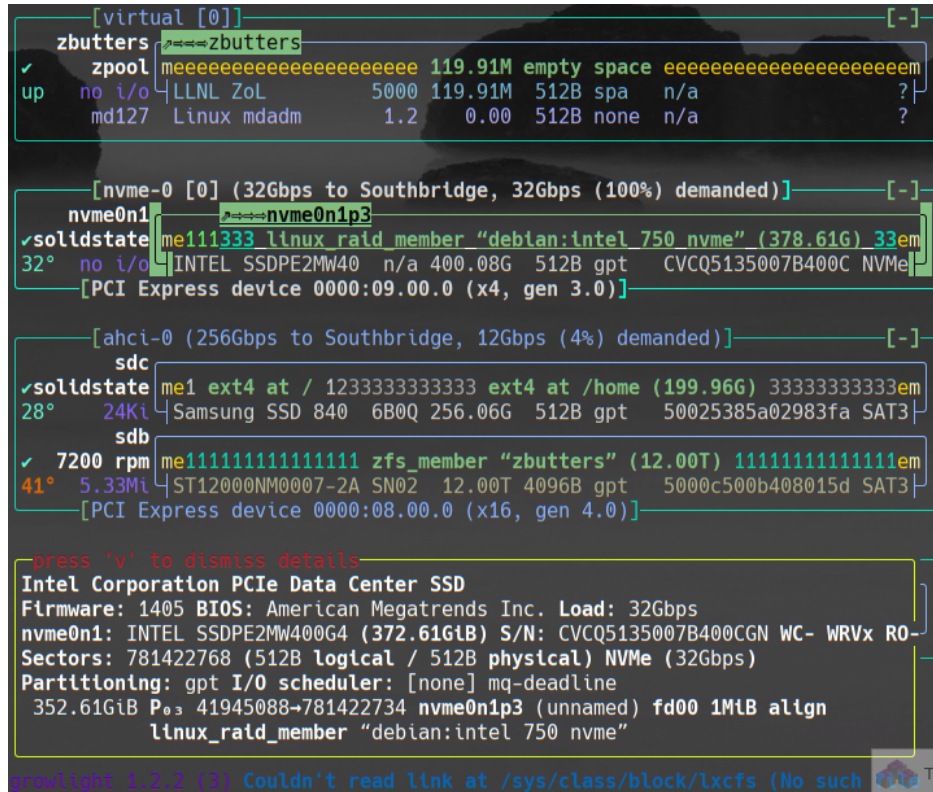


FIGURE 53: growlight, a program built around reels.

```

int ncselector_additem(struct ncselector* n, const struct selector_item* item);
int ncselector_delitem(struct ncselector* n, const char* item);

// Return a reference to the selected option, or NULL if there are no items.
const char* ncselector_selected(const struct ncselector* n);

// Return a reference to the ncselector's underlying ncplane.
struct ncplane* ncselector_plane(struct ncselector* n);

// Move up or down in the list. A reference to the newly-selected item is
// returned, or NULL if there are no items in the list.
const char* ncselector_previtem(struct ncselector* n);
const char* ncselector_nextitem(struct ncselector* n);

// Offer the input to the ncselector. If it's relevant, this function returns
// true, and the input ought not be processed further. If it's irrelevant to
// the selector, false is returned. Relevant inputs include:
// * a mouse click on an item
// * a mouse scrollwheel event
// * a mouse click on the scrolling arrows
// * a mouse click outside of an unrolled menu (the menu is rolled up)
// * up, down, pgup, or pgdown on an unrolled menu (navigates among items)
bool ncselector_offer_input(struct ncselector* n, const struct ncinput* nc);

// Destroy the ncselector. If 'item' is not NULL, the last selected option will
// be strdup()ed and assigned to '*item' (and must be free()d by the caller).
void ncselector_destroy(struct ncselector* n, char** item);

```

LISTING 85: Selector control.

```

typedef struct menu_options {
    bool bottom;           // on the bottom row, as opposed to top row
    bool hiding;          // hide the menu when not being used
    struct {
        char* name;       // utf-8 c string
        struct {
            char* desc;   // utf-8 menu item, NULL for horizontal separator
            ncinput shortcut; // shortcut, all should be distinct
        } items;
        int itemcount;
    } * sections;         // array of menu sections
    int sectioncount;    // must be positive
    uint64_t headerchannels; // styling for header
    uint64_t sectionchannels; // styling for sections
} menu_options;

struct ncmenu;

// Create a menu with the specified options. Menus are currently bound to an overall notcurses object
// (as opposed to a particular plane), and are implemented as ncplanes kept atop other ncplanes.
struct ncmenu* ncmenu_create(struct notcurses* nc, const menu_options* opts);

```

LISTING 86: Menu creation.


```

// Unroll the specified menu section, making the menu visible if it was invisible, and rolling
// up any menu section that is already unrolled.
int ncmenu_unroll(struct ncmenu* n, int sectionidx);

// Roll up any unrolled menu section, and hide the menu if using hiding.
int ncmenu_rollback(struct ncmenu* n);

// Return the selected item description, or NULL if no section is unrolled. If 'ni' is not NULL,
// and the selected item has a shortcut, 'ni' will be filled in with that shortcut. This can allow faster matching.
const char* ncmenu_selected(const struct ncmenu* n, struct ncinput* ni);

// Return the ncplane backing this ncmenu.
struct ncplane* ncmenu_plane(struct ncmenu* n);

// Offer the input to the ncmenu. If it's relevant, this function returns true, and the input ought not be
// processed further. If it's irrelevant to the menu, false is returned. Relevant inputs include:
// * mouse movement over a hidden menu
// * a mouse click on a menu section (the section is unrolled)
// * a mouse click outside of an unrolled menu (the menu is rolled up)
// * left or right on an unrolled menu (navigates among sections)
// * up or down on an unrolled menu (navigates among items)
// * escape on an unrolled menu (the menu is rolled up)
bool ncmenu_offer_input(struct ncmenu* n, const struct ncinput* nc);

// Destroy a menu created with ncmenu_create().
int ncmenu_destroy(struct ncmenu* n);

```

LISTING 87: Menu control.

placed in the center). Further ncreel functionality is detailed in Listing 90.

Several widgets will probably be added, possibly before Notcurses 2.0:

- A libreadline wrapper, or equivalent functionality.
- Histogram and line plot capabilities using block elements and Braille.
- A HUD for direct mode.

13.4 Example: let's rip off whiptail

The colloquy program shipped with Notcurses implements a command-line API similar to that of the Newt program whiptail, which itself ripped off the NCURSES program dialog. All of these programs allow simple user interfaces to be thrown up on the command line. colloquy is written in Rust⁸⁶, using the Notcurses crate⁸⁷, itself a wrapper of the libnotcurses-sys⁸⁸ bindgen-generated Rust wrappers.

⁸⁶<https://lib.rs/crates/colloquy>

⁸⁷<https://lib.rs/crates/notcurses>

⁸⁸<https://lib.rs/crates/libnotcurses-sys>

```

typedef struct ncreel_options {
    // require this many rows and columns (including borders). otherwise, a message will be displayed stating
    // that a larger terminal is necessary, and input will be queued. if 0, no minimum will be enforced. may
    // not be negative. note that ncreel_create() does not return error if given a plane smaller than these
    // minima; it instead patiently waits for the screen to get bigger.
    int min_supported_cols;
    int min_supported_rows;

    // use no more than this many rows and columns (including borders). may not be less than the
    // corresponding minimum. 0 means no maximum.
    int max_supported_cols;
    int max_supported_rows;

    // desired offsets within the surrounding WINDOW (top right bottom left) upon creation / resize. an
    // ncreel_move() operation updates these.
    int toff, roff, boff, loff;
    // is scrolling infinite (can one move down or up forever, or is an end reached?). if true, 'circular'
    // specifies how to handle the special case of an incompletely-filled reel.
    bool infinitescroll;
    // is navigation circular (does moving down from the last tablet move to the first, and vice versa)?
    // only meaningful when infinitescroll is true. if infinitescroll is false, this must be false.
    bool circular;
    // notcurses can draw a border around the ncreel, and also around the component tablets. inhibit
    // borders by setting all valid bits in the masks. partially inhibit borders by setting individual
    // bits in the masks. the appropriate attr and pair values will be used to style the borders. focused
    // and non-focused tablets can have different styles. you can instead draw your own borders, or
    // forgo borders entirely.
    unsigned bordermask; // bitfield; 1s will not be drawn (see bordermaskbits)
    uint64_t borderchan; // attributes used for ncreel border
    unsigned tabletmask; // bitfield; same as bordermask but for tablet borders
    uint64_t tabletchan; // tablet border styling channel
    uint64_t focusedchan; // focused tablet border styling channel
    uint64_t bgchannel; // background colors
} ncreel_options;

struct ncreel;

// Create an ncreel according to the provided specifications. Returns NULL on failure. 'nc' must be a
// valid plane, to which offsets are relative. Note that there might not be enough room for the
// specified offsets, in which case the ncreel will be clipped on the bottom and right. A minimum number
// of rows and columns can be enforced via popts. efd, if non-negative, is an eventfd/pipe that ought be
// written to whenever ncreel_touch() updates a tablet (this is useful in the case of nonblocking input).
struct ncreel* ncreel_create(struct ncplane* nc, const ncreel_options* popts, int efd);

```

LISTING 88: Reel creation.

```
// Tablet draw callback, provided a tablet (from which the ncpplane and userptr may be extracted),
// the first column that may be used, the first row that may be used, the first column that may not
// be used, the first row that may not be used, and a bool indicating whether output ought be
// clipped at the top (true) or bottom (false). Rows and columns are zero-indexed, and both are
// relative to the tablet's plane.
//
// Regarding clipping: it is possible that the tablet is only partially displayed on the screen. If
// so, it is either partially present on the top of the screen, or partially present at the bottom.
// In the former case, the top is clipped (cliptop will be true), and output ought start from the
// end. In the latter case, cliptop is false, and output ought start from the beginning.
//
// Returns the number of lines of output, which ought be less than or equal to
// maxy - begy, and non-negative (negative values might be used in the future).
typedef int (*tabletc_b)(struct ncpplane* t, int begx, int begy, int maxx, int maxy, bool cliptop);
```

LISTING 89: Tablet redraw callback function type.

```
// Returns the ncplane on which this ncreel lives.
struct ncplane* ncreel_plane(struct ncreel* pr);

// Add a new tablet to the provided ncreel, having the callback object opaque. Neither, either, or both of
// after and before may be specified. If neither is specified, the new tablet can be added anywhere on the
// reel. If one or the other is specified, the tablet will be added before or after the specified tablet.
// If both are specified, the tablet will be added to the resulting location, assuming it is valid
// (after->next == before->prev); if it is not valid, or there is any other error, NULL will be returned.
struct nctablet* ncreel_add(struct ncreel* pr, struct nctablet* after, struct nctablet* before, tabletcb cb, void* opaque);

// Return the number of tablets.
int ncreel_tabletcount(const struct ncreel* pr);

// Indicate that the specified tablet has been updated in a way that would change its display.
// This will trigger some non-negative number of callbacks (though not in the caller's context).
int ncreel_touch(struct ncreel* pr, struct nctablet* t);

// Delete the tablet specified by t from the ncreel specified by pr. Returns -1 if the tablet cannot be found.
int ncreel_del(struct ncreel* pr, struct nctablet* t);

// Delete the active tablet. Returns -1 if there are no tablets.
int ncreel_del_focused(struct ncreel* pr);

// Move to the specified location within the containing plane.
int ncreel_move(struct ncreel* pr, int x, int y);

// Redraw the ncreel in its entirety, for instance after
// clearing the screen due to external corruption, or a SIGWINCH.
int ncreel_redraw(struct ncreel* pr);

// Return the focused tablet, if any tablets are present. This is not a copy;
// be careful to use it only for the duration of a critical section.
struct nctablet* ncreel_focused(struct ncreel* pr);

// Change focus to the next tablet, if one exists
struct nctablet* ncreel_next(struct ncreel* pr);

// Change focus to the previous tablet, if one exists
struct nctablet* ncreel_prev(struct ncreel* pr);

// Destroy an ncreel allocated with ncreel_create(). Does not destroy the
// underlying plane. Returns non-zero on failure.
int ncreel_destroy(struct ncreel* pr);

// Returns a pointer to a user pointer associated with this nctablet.
void* nctablet_userptr(struct nctablet* t);

// Access the ncplane associated with this tablet, if one exists.
struct ncplane* nctablet_ncplane(struct nctablet* t);
```

LISTING 90: Reel control.

14 Complex examples

14.1 Example: walking through notcurses-demo

The `notcurses-demo` program is built as part of Notcurses, and ought have been installed alongside the library (on Debian, you'll need the `notcurses-bin` package, and even then the demo has been somewhat reduced in order to comply with the DFSG[35]). It demonstrates a wide range of Notcurses capabilities, and its source code is most instructive.

It is best to run the demo in a terminal having geometry of at least 80x45, though anything 80x24 or larger will more or less work (some content will be clipped). It is also desirable to have 24-bit color enabled, assuming your terminal supports it. Determine the number of colors advertised by your terminal type using `infocmp` (see Figure 54). Some relevant terminfo capabilities are described in Table 13.

colors	Integer	Number of colors.
ccc	Boolean	The palette can be programmed.
RGB	Boolean	Direct RGB values can be specified.

TABLE 13: Relevant terminfo properties.

Each demo makes use of a few different Notcurses capabilities. In addition, a menu is present throughout. From this menu (or using keyboard shortcuts), you can activate a HUD (H) and an informational help display (Ctrl+u). In addition, you can restart the demo with Ctrl+R, or quit at any time (q). This application serves admirably for benchmarking certain terminal behaviors, and we'll do exactly that in Appendix B. The performance properties of various components are described at length therein.

```
[schwarzgerat](0) $ echo $TERM
xterm-256color
[schwarzgerat](0) $ infocmp | egrep -e ccc\|colors\|RGB
xterm-256color|xterm with 256 colors,
  am, bce, ccc, km, mc5i, mir, msgr, npc, xenl,
  colors#0x100, cols#80, it#8, lines#24, pairs#0x10000,
[schwarzgerat](0) $ export TERM=xterm-direct
[schwarzgerat](0) $ infocmp | egrep -e ccc\|colors\|RGB
  colors#0x1000000, cols#80, it#8, lines#24, pairs#0x10000,
[schwarzgerat](0) $ export TERM=xterm
[schwarzgerat](0) $ infocmp | egrep -e ccc\|colors\|RGB
  colors#8, cols#80, it#8, lines#24, pairs#64,
[schwarzgerat](0) $ export TERM=alacritty
[schwarzgerat](0) $ infocmp | egrep -e ccc\|colors\|RGB
  am, bce, ccc, hs, km, mc5i, mir, msgr, npc, xenl,
  colors#0x100, cols#80, it#8, lines#24, pairs#0x10000,
[schwarzgerat](0) $ export TERM=alacritty-direct
[schwarzgerat](0) $ infocmp | egrep -e ccc\|colors\|RGB
  colors#0x1000000, cols#80, it#8, lines#24, pairs#0x10000,
[schwarzgerat](0) $
```

FIGURE 54: Inspecting the terminfo database.

Screenshots were taken using `scrot` 1.2 and a 80x45 `xfce4-terminal` 0.8.9.1 from Xfce 4.14+Compiz 0.8.16.1 atop Xorg 1.20.7 on NVIDIA 440.59. All of these are the unmodified Debian Unstable x86_64 binaries. My kernel is a custom 5.5.6 build. The terminal type is `vte-256color`, and `COLORTERM` is defined to be 24bit. The terminal font was Hack 10, and the background is a 0.7 transparency.

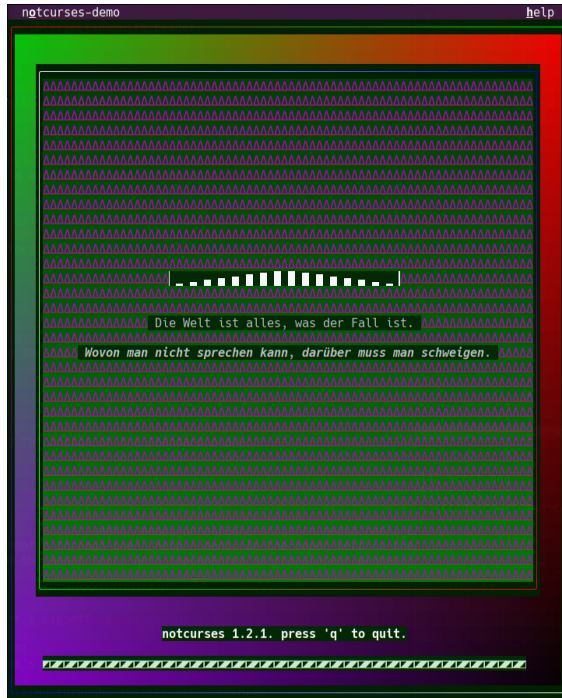


FIGURE 55: “Intro”. Lerps on the perimeters. Inverse radial gradient plus vertical gradient. Full-screen fade. Cyclic glyphs. Italics.



FIGURE 56: “X-Ray”. Streaming video. Very large planes (the scrolling plane at the bottom is much larger than the visible screen).

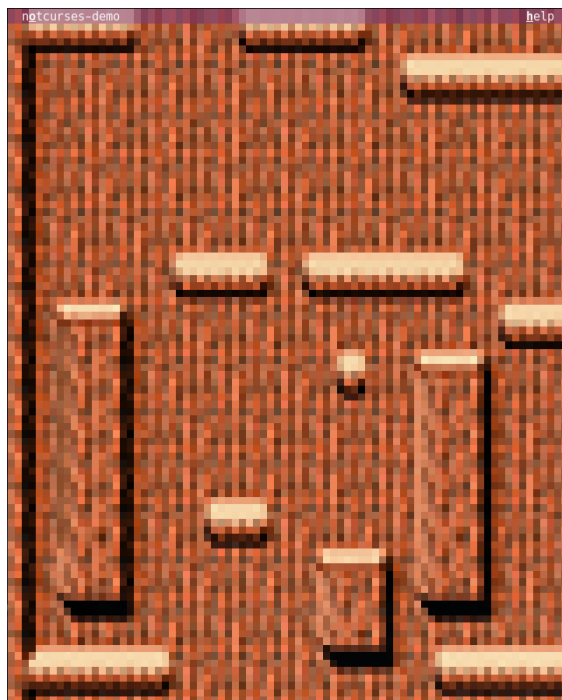


FIGURE 57: “Eagle”, first phase. Parallax scrolling on large image.

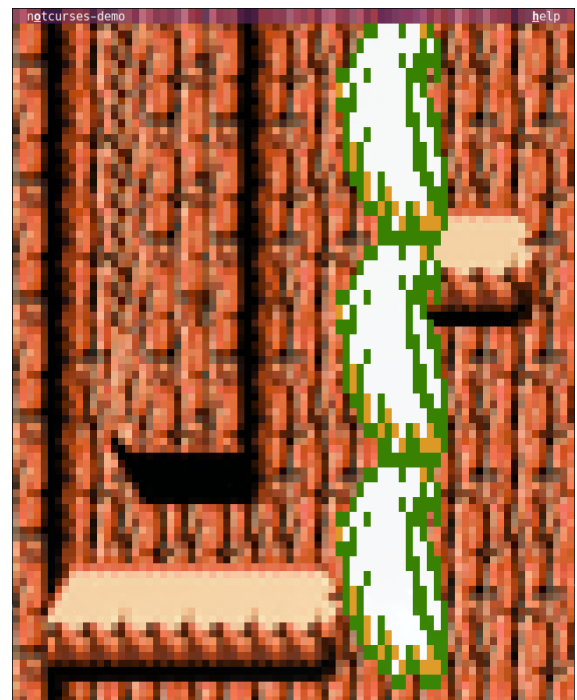


FIGURE 58: “Eagle”, second phase. Sprites. Zoomed image.



FIGURE 59: “Trans”. Transparent top plane. Window through to the desktop.



FIGURE 60: “Trans”. Opaque foreground, transparent background, no glyph.

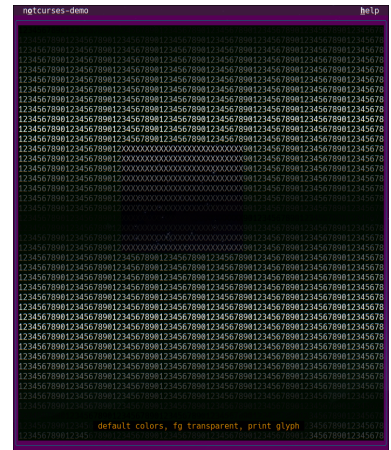


FIGURE 61: “Trans”. Transparent foreground and background with opaque glyph.

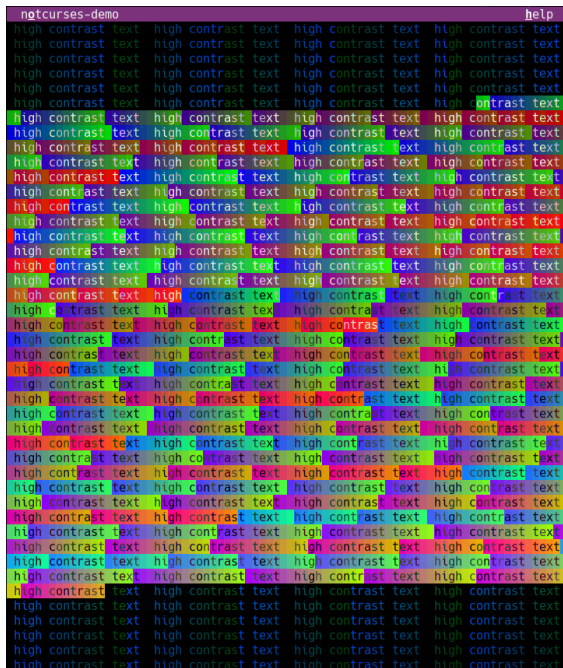


FIGURE 62: “Highcon”. High-contrast text.

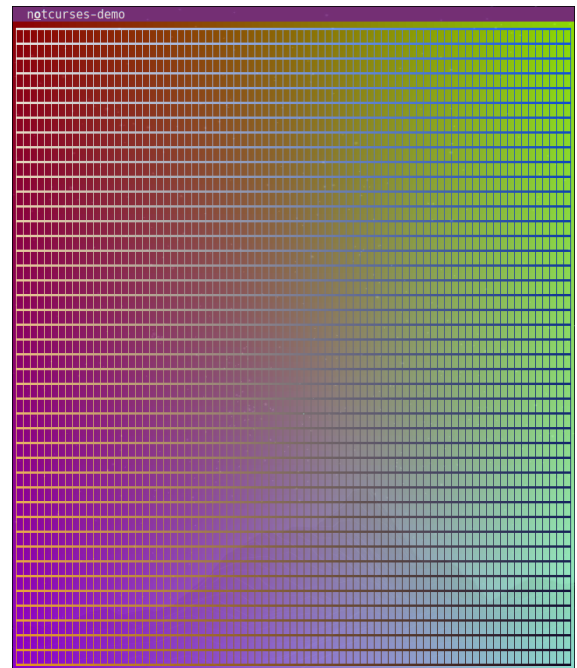


FIGURE 63: “Grid”. Max RGB density.

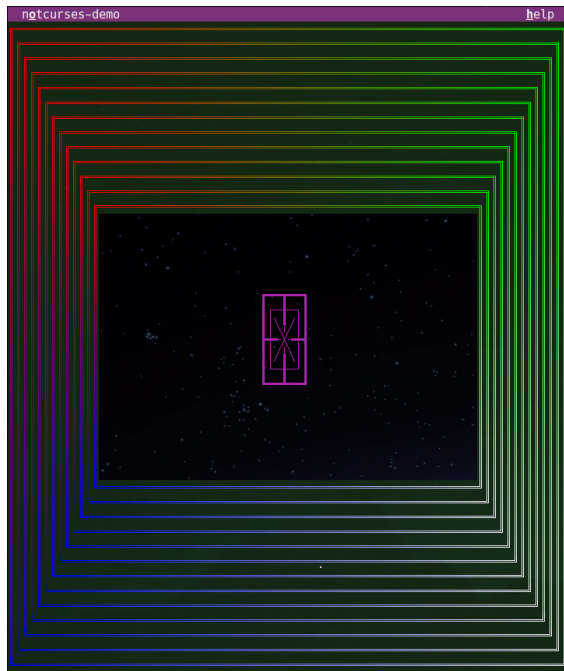


FIGURE 64: “Box”. Lerped perimeters. Precise Unicode. Color sweeps.

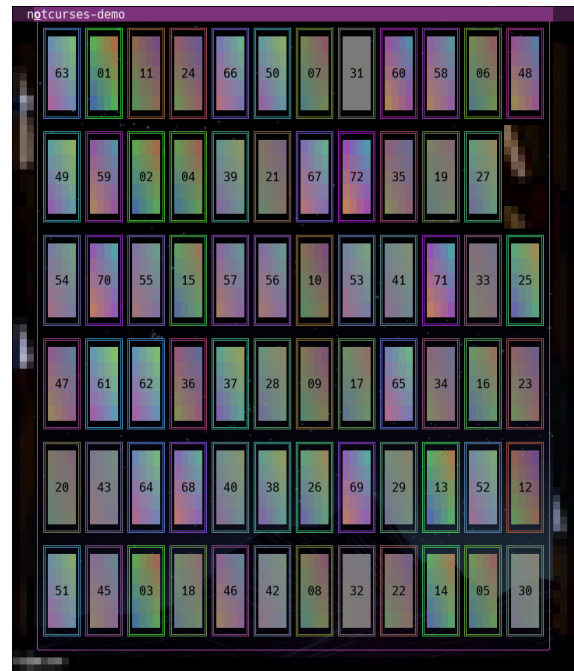


FIGURE 65: “Sliders”. Partial fades. Animation. Gradients.

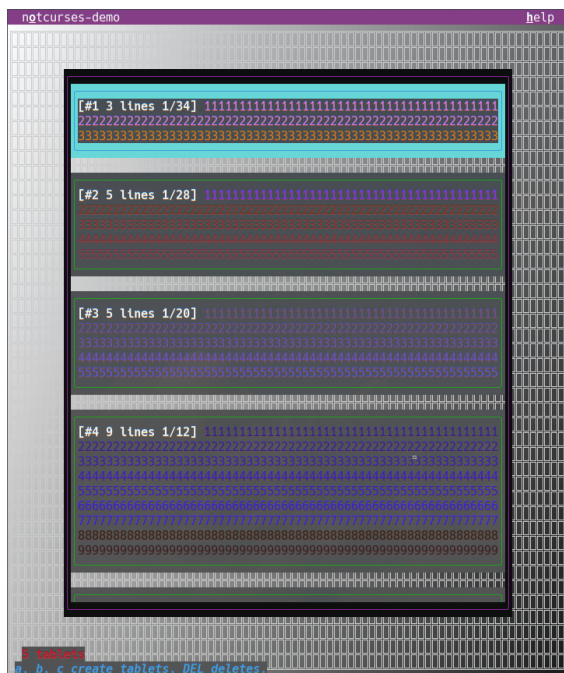


FIGURE 66: “Reels”. The ncreel widget.

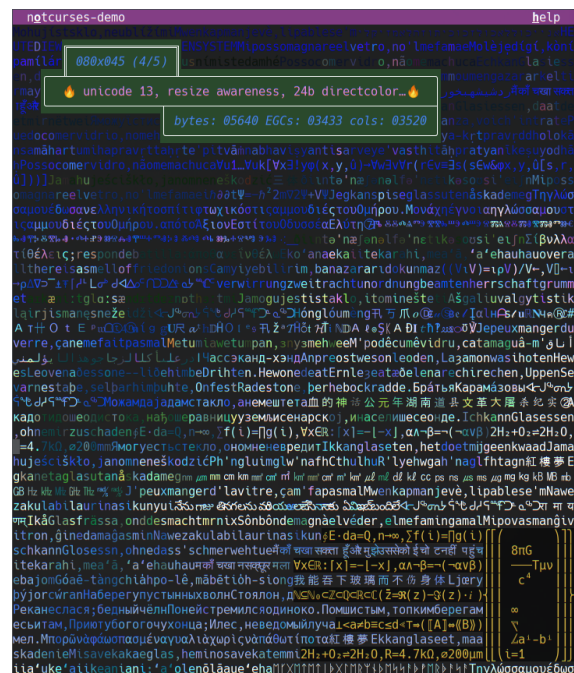


FIGURE 67: “Whiteout”. Translucency.

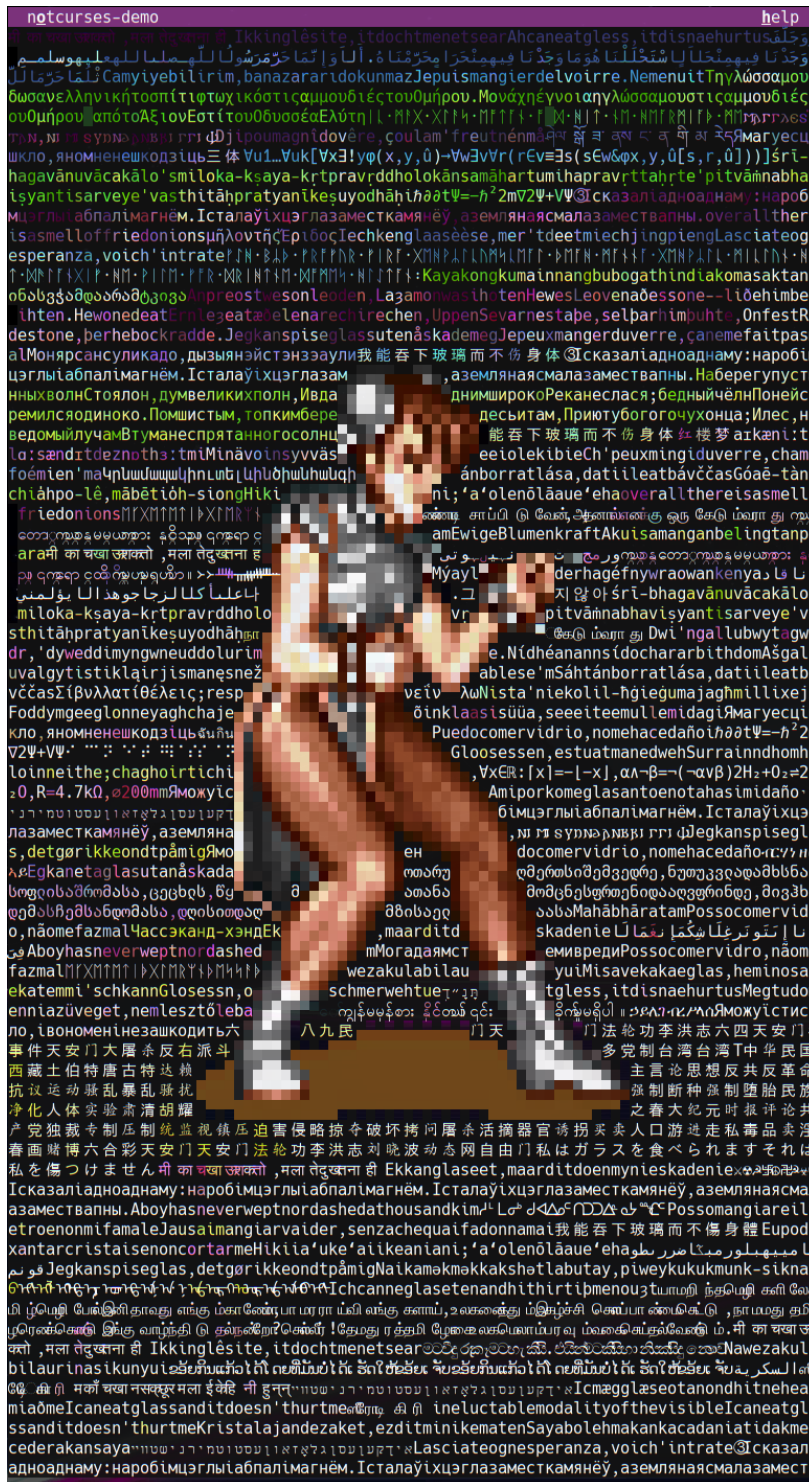


FIGURE 68: “Chunli”. Sprite animation.

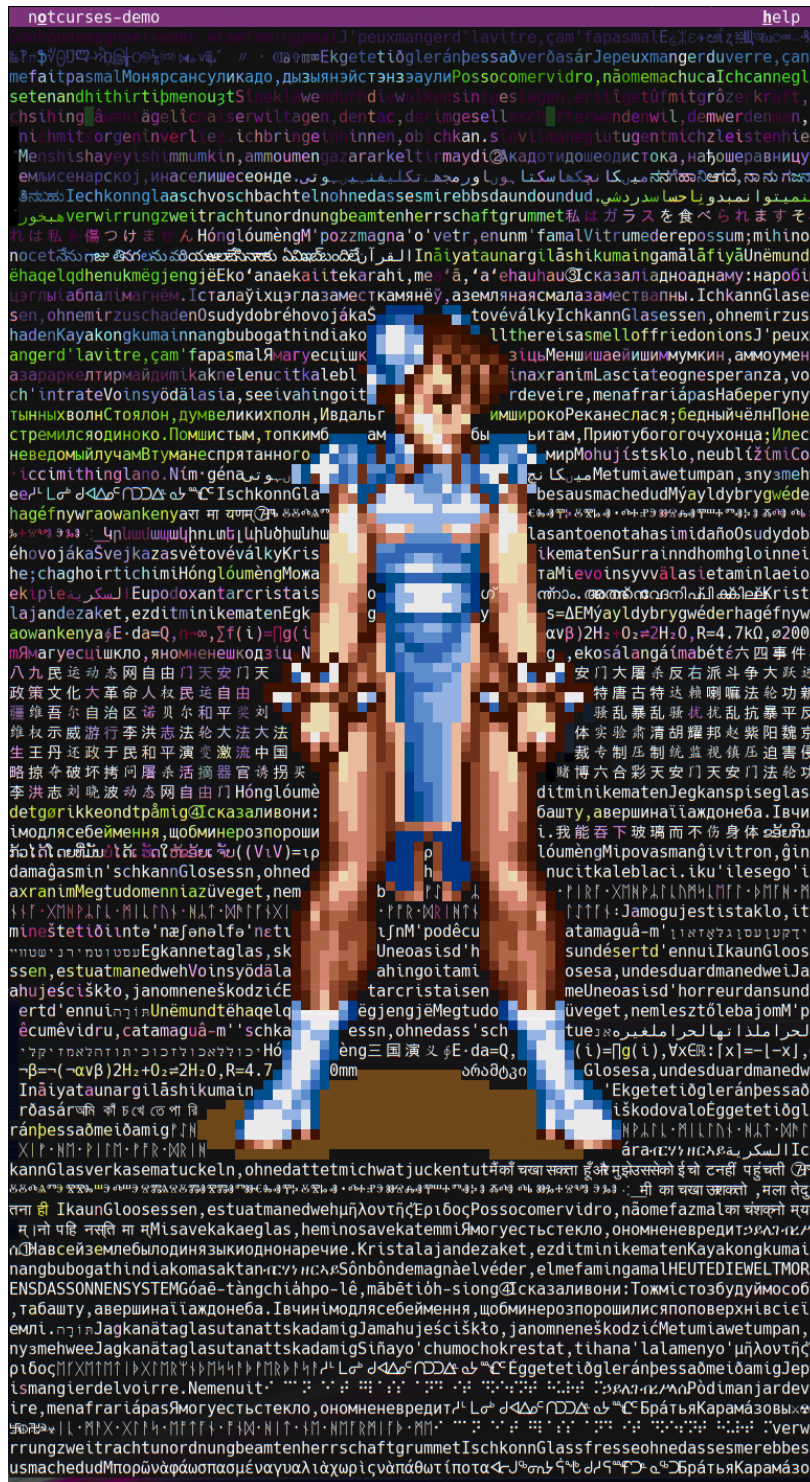


FIGURE 69: “Chunli”. Sprite animation.



FIGURE 70: “Uniblock”. Hangul syllables.



FIGURE 71: “Uniblock”. Emoji.

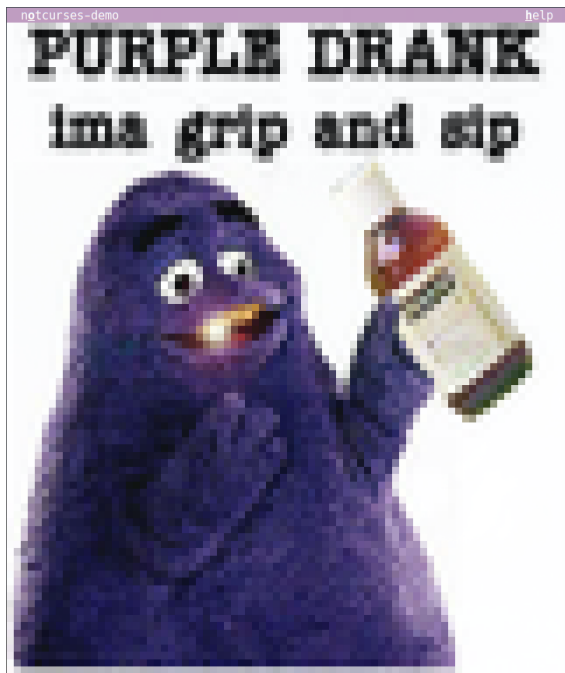


FIGURE 72: “View”. Scaling an image.

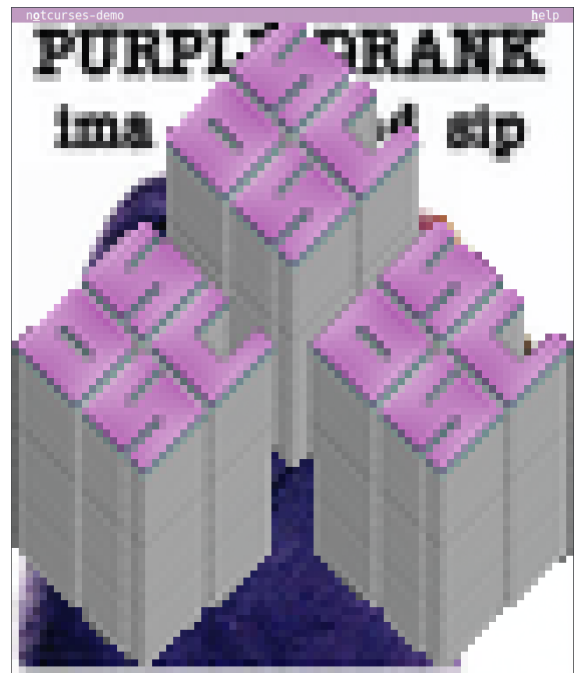


FIGURE 73: “View”. Transparent images.

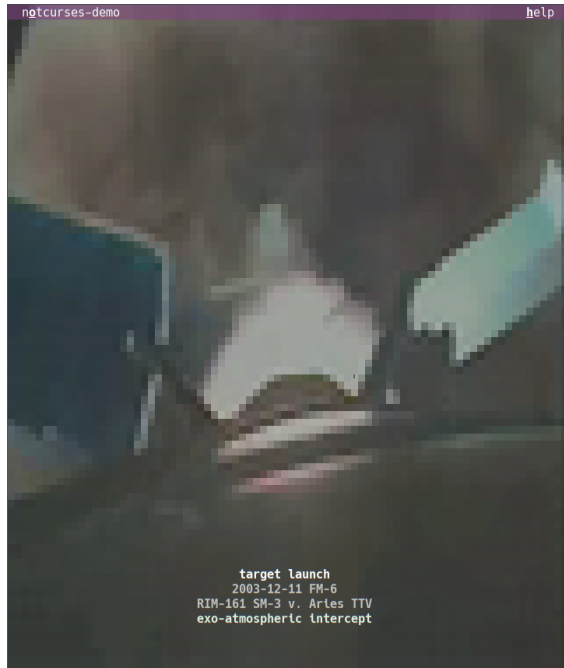


FIGURE 74: “View”. Streaming video with high-contrast text.



FIGURE 75: “View”. Notice the high-contrast kicking in.

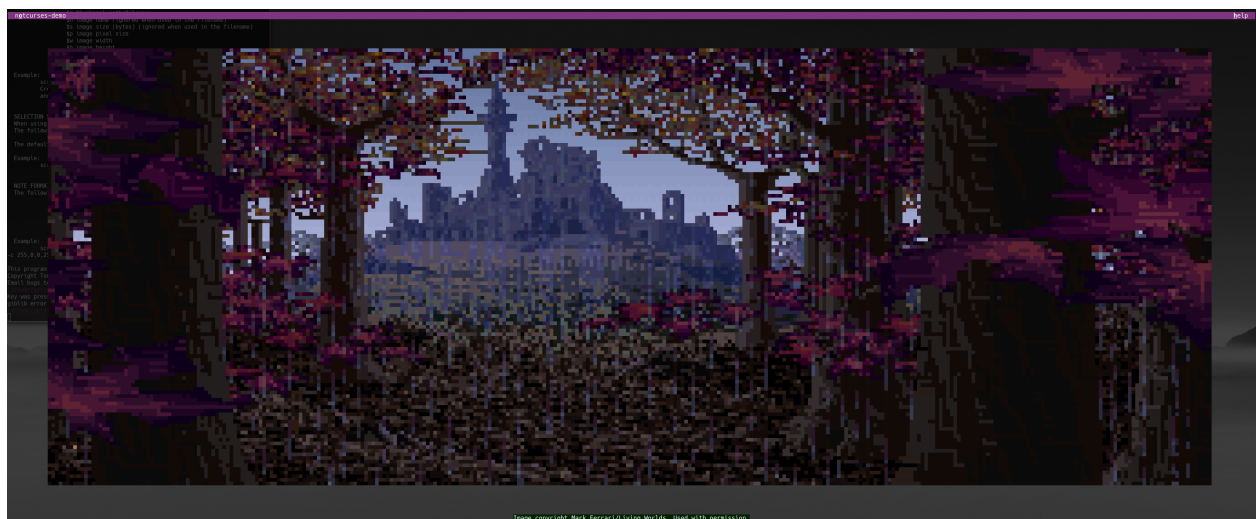


FIGURE 76: “Jungle”. Palette-indexed image. Very low-bandwidth animation via palette cycling. “Ruins in Rain” © Mark Ferrari/Living Worlds. Texelized with permission.

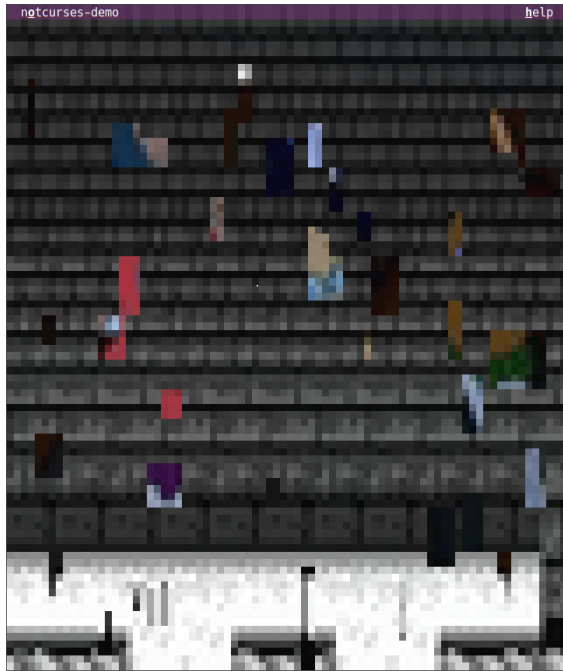


FIGURE 77: “Fallin”. Color change, introspection, many planes.



FIGURE 78: “Fallin”. The underlying image is revealed.

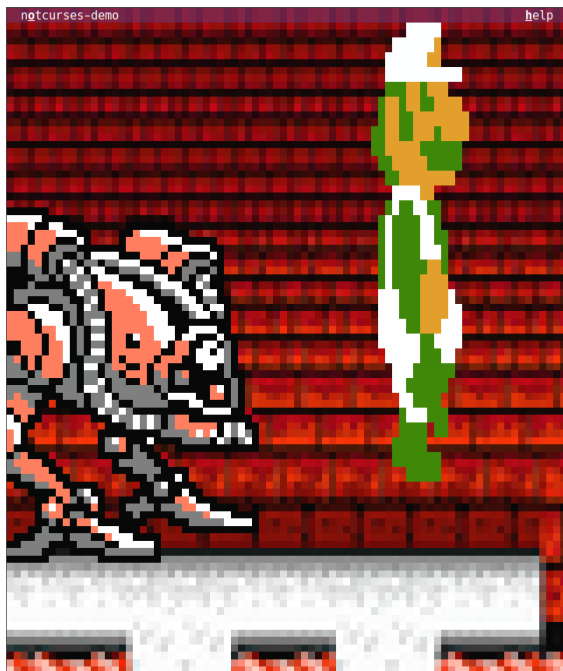


FIGURE 79: “Luigi”. Multiple sprites.

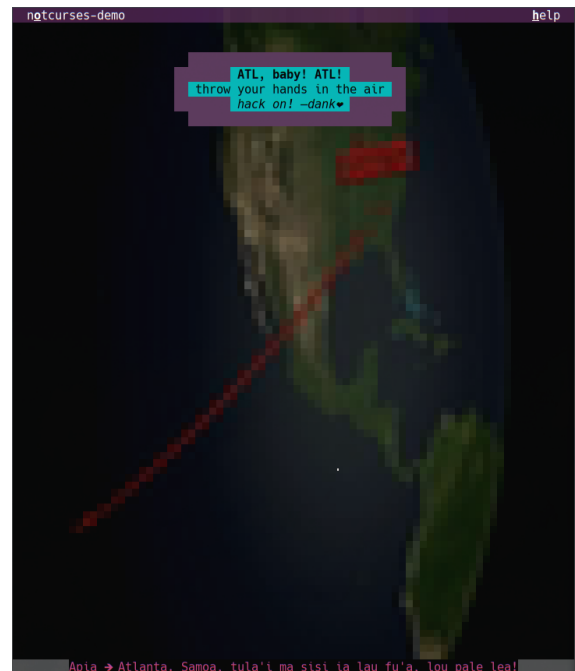


FIGURE 80: “Outro”. Fades atop video.

14.2 Example: let's rip off tetris

Recall our exploration of tetriminos from Chapter 5. We know enough now to turn this into an actual game of terminal Tetris⁸⁹. Our implementation is fewer than 200 lines total, yet it unites most of the techniques introduced in the past few chapters. For fun, we'll do this in C++⁹⁰, using Marek Habersack's C++ wrappers (installed with Notcurses).

Our `main()` will parse command line options, set up a `Tetris` object appropriately, and watch for input. The `Tetris` object provides a thread function `Ticker` which moves the current piece down according to timer events (calling `StuckPiece` to determine if the piece has been placed, in which case a new piece enters the playing area), and the necessary interface for the input loop:

- `MoveLeft()` and `MoveRight()` (Listing 83). These simply verify that the lateral move can be done, and then call `Plane::move()`.
- `MoveDown()` (Listing 95), which calls the same `StuckPiece()`,
- `Pause()`, and
- `RotateCCW()` and `RotateCW()`, which verify that the rotation is possible, and then call `Plane::rotate_cw()` or `Plane::rotate_ccw()`.

The general structure of our solution is thus:

- A background is drawn onto the standard plane.
- A new plane is created for the game area. Why make a new plane? Recall the beginning of Chapter 8: *we use a plane wherever we benefit from distinct state*. By keeping the game area on its own plane, we can trivially move it in response to terminal resizes, and likewise trivially test whether the current piece can move in a given direction.
- Two tetrimino planes are created, one for the current piece, and one for the next piece. The current piece descends from the top of the game area. Upon reaching its final position, its plane is added to the game plane using `ncplane_mergedown()`. The next piece is brought to the top of the game area, and the current piece is redrawn on its way to becoming the next piece.
- We need no external state save the score and the order of pieces. The order of pieces is left up to the PRNG. We only need track the score and our four planes. The validity of a given movement can be checked entirely by reflection, using `ncplane_at_yx()`.
- Our main thread loops on `Notcurses::getc()`, calling into the `Tetris` object. These calls will need to lock against the timer thread.
- If `StuckPiece()` returns `true`, and the stuck piece is at the top of the playing area, the game is over.

The playing area should not be visible while the game is paused, so whenever the game is paused, we'll move the standard plane to the top of the z-axis. As it is opaque and spans the visible area, this will hide the playing area. We'll furthermore throw up a three-row plane centered on the screen; this plane will contain a perimeter, and pulsing text reading "Paused". On an unpaue event, the attract plane is destroyed, and the standard plane moved back to the bottom of the z-axis. Nothing else needs be touched, save inhibiting the operation of `Ticker()`. This could be done any number of ways; we'll use a condition variable plus a bit flag, checking it upon emerging from our timeout.

Our resize logic is pretty simple: on a resize event, after verifying that the new geometry is large enough to play on (if not, we pause the game), we must redraw the background, move the playing area to the new bottom center of the visible area, and move the current piece to its same location relative to the playing area. If the game is already paused when we resize, we ought additionally recenter the attract plane⁹¹.

⁸⁹There are a great many distinct Tetris implementations. We'll aim for loose conformance to the NES version as published by Nintendo of America, because I'm old. See [https://tetris.wiki/Tetris_\(NES,_Nintendo\)](https://tetris.wiki/Tetris_(NES,_Nintendo)). I'm not going to obsess over details, though.

⁹⁰LOL how often does one hear that said?

⁹¹Imagine that we sized the playing area according to the visible area. This would require resizing the game board upon a terminal resize, a decidedly messier affair. It's *doable*—ἦ τὰν ἦ ἐπὶ τὰς.

```

static constexpr int MAX_LEVEL = 15;
// the number of milliseconds before a drop is forced at the given level,
// using the NES fps counter of 50ms
static constexpr int Gravity(int level) {
    constexpr int MS_PER_GRAV = 30; // 10MHz*63/88/455/525 (~29.97fps) in NTSC
    // The number of frames before a drop is forced, per level
    constexpr std::array<int, MAX_LEVEL + 1> Gravities = {
        43, 38, 33, 28, 23, 18, 13, 8, 6, 5, 5, 4, 4, 3, 2, 1
    };
    if(level < 0){
        throw std::out_of_range("Illegal level");
    }
    if(static_cast<unsigned long>(level) < Gravities.size()){
        return Gravities[level] * MS_PER_GRAV;
    }
    return MS_PER_GRAV; // all levels 29+ are a single grav
}

void StainBoard(int dimy, int dimx){
    if(!board->cursor_move(0, 1)){
        throw TetrisNotcursesErr("cursor_move()");
    }
    int high = 0xff - level_ * 16, low = level_ * 16; // rgb calculation limits us to 16 levels (0--15)
    uint64_t tl = 0, tr = 0, bl = 0, br = 0;
    channels_set_fg_rgb(&tl, high, 0xff, low); channels_set_bg_alpha(&tl, CELL_ALPHA_TRANSPARENT);
    channels_set_fg_rgb(&tr, low, high, 0xff); channels_set_bg_alpha(&tr, CELL_ALPHA_TRANSPARENT);
    channels_set_fg_rgb(&bl, 0xff, low, high); channels_set_bg_alpha(&bl, CELL_ALPHA_TRANSPARENT);
    channels_set_fg_rgb(&br, 0xff, high, low); channels_set_bg_alpha(&br, CELL_ALPHA_TRANSPARENT);
    if(!board->stain(dimy - 2, dimx - 2, tl, tr, bl, br)){
        throw TetrisNotcursesErr("stain()");
    }
}
}

```

LISTING 91: Tetris helpers Gravity() and StainBoard().

Let's do some boring groundwork first. We'll need the official constants for "gravity", the level-dependent rate at which pieces fall (Listing 91). We can implement this as a `constexpr` function, not that it's likely to be of any real advantage (especially since we only call `Gravity()` upon level changes).

```

// returns true iff the specified row of the board is clear (full with no gaps)
bool LineClear(int y){
    int dimx = board->get_dim_x();
    for(int x = 1 ; x < dimx - 1 ; ++x){
        ncpp::Cell c;
        if(board->get_at(y, x, &c) < 0){
            throw TetrisNotcursesErr("get_at()");
        }
        if(c.is_simple()){
            return false;
        }
    }
    return true;
}
}

```

LISTING 92: Tetris::LineClear().

We'll need a function to tell us whether a line has been cleared (Listing 92). We could track the state of the

board ourselves as a simple boolean matrix, but why bother when we can just ask the source? Reflecting on a Notcurses plane is good practice of the DRY⁹² principle. These aren't system calls, just cheap indexed lookups into a framebuffer. It's unlikely that you can do significantly better with your own implementation, so why bother? Let Notcurses handle the state for you.

Our background function (Listing 93) just loads up the provided image, stretched to fill the standard plane, blits it, and then converts it to greyscale (so as not to be confused with actual playing pieces, all of which are in color). The board is a double-lined box with the top missing, and is its own plane. We also make a distinct plane for the score and other textual info; this is useful as a last-ditch stopgap preventing such text from spilling into the play area. Placing the board on its own plane (as opposed to blitting it destructively onto the background) has two major advantages: it simplifies responding to a terminal resize, and it allows us to determine illegal moves via reflection on this plane.

This brings us to `InvalidMove()` (Listing 94). All of our movement functions to come will need a means to test whether the selected movement is legal, whether it's a lateral translation (`MoveLeft()` and `MoveRight()`), a rotation (`RotateCw()` and `RotateCcw()`), or falling towards the bottom (`MoveDown()`, called when the user presses down and when the timer expires). Similarly to `LineClear()`, it's easiest to leverage the Notcurses state. All five of these functions operate by speculatively transforming the current piece, testing for overlap with the gameboard plane, and pulling the piece back if there was an overlap.

As mentioned above, `MoveDown()` is called both by the timer thread, and by the UI when the user initiates a drop. `MoveDown()` is special among the movement functions: whereas the others undo an invalid move, `MoveDown()` recognizes such as the current piece having bottomed out. If the location is above the gameboard, the game is over. Otherwise, we call `LockPiece()` to fuse the current piece with the stack, and bring a new piece into play.

The timer is managed in `Ticker()` (Listing 96), which is run inside its own thread.

When the current piece reaches its resting place, `LockPiece()` (Listing 97) is called. This merges the current piece down onto the playing area⁹³, scans for any cleared lines, removes them, allows the material above these lines to fall, and finally repaints a gradient onto the stack before rendering. The gradient slowly changes over different levels—indeed, the gradient's computation limits us to 16 levels.

`NewPiece()` (Listing 98) is called at the beginning of the game, and whenever a piece is locked in. We saw most of this already in Chapter 5.

Our `main()` is quite minimal. A `Tetris` instance is constructed, initializing the game board and providing necessary state (score, level, etc.). `Tetris::Ticker()` is launched as a C++11 thread. We then loop on input until either the game is over or the user presses 'q'. Finally, we `join()` the `Ticker()` thread, and shut down Notcurses. We accept function keys or vi keys for movement. 'z' and 'x' rotate counterclockwise and clockwise respectively. Ctrl+L refreshes the screen to round out our UI.

Note that, once again, a majority of our actual lines of code are devoted to the detection and propagation of errors. Our C++ wrappers do not throw exceptions themselves; I will likely add a further C++ wrapper which does in the future. Should that come into play, all these error checks and manual `throw` directives could be sweetly elided.

⁹²Don't Repeat Yourself.

⁹³The "stack", at least in Tetris parlance.

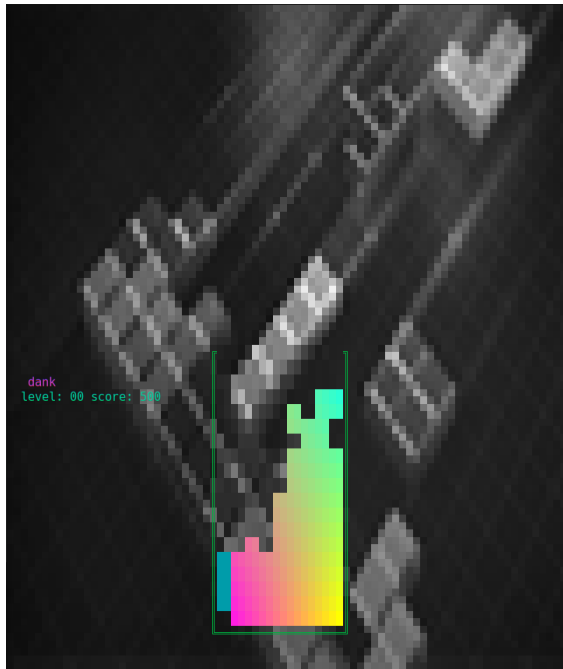


FIGURE 81: Tetris—primed to score.

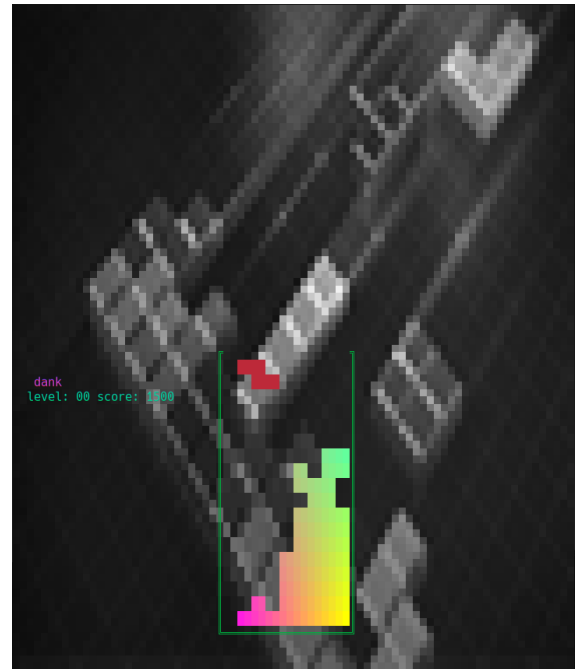


FIGURE 82: Tetris—boom!

```

void MoveLeft() {
    const std::lock_guard<std::mutex> lock(mtx_);
    int y, x;
    if(PrepForMove(&y, &x)){
        if(!curpiece_>move(y, x - 2)){
            throw TetrisNotcursesErr("move()");
        }
        if(InvalidMove()){
            if(!curpiece_>move(y, x)){
                throw TetrisNotcursesErr("move()");
            }
        }else{
            x -= 2;
            if(!Inc_.render()){
                throw TetrisNotcursesErr("render()");
            }
        }
    }
}

```

```

void MoveRight() {
    const std::lock_guard<std::mutex> lock(mtx_);
    int y, x;
    if(PrepForMove(&y, &x)){
        if(!curpiece_>move(y, x + 2)){
            throw TetrisNotcursesErr("move()");
        }
        if(InvalidMove()){
            if(!curpiece_>move(y, x)){
                throw TetrisNotcursesErr("move()");
            }
        }else{
            x += 2;
            if(!Inc_.render()){
                throw TetrisNotcursesErr("render()");
            }
        }
    }
}

```

FIGURE 83: Tetris::MoveRight() and Tetris::MoveLeft().

```
void RotateCw() {
    const std::lock_guard<std::mutex> lock(mtx_);
    int y, x;
    if(!PrepForMove(&y, &x)){
        return;
    }
    if(!curpiece_>rotate_cw()){
        throw TetrismotcursesErr("rotate_cw()");
    }
    if(InvalidMove() && !curpiece_>rotate_ccw()){
        throw TetrismotcursesErr("rotate_ccw()");
    }
    if(!nc_.render()){
        throw TetrismotcursesErr("render()");
    }
}

void RotateCcw() {
    const std::lock_guard<std::mutex> lock(mtx_);
    int y, x;
    if(!PrepForMove(&y, &x)){
        return;
    }
    if(!curpiece_>rotate_ccw()){
        throw TetrismotcursesErr("rotate_ccw()");
    }
    if(InvalidMove() && !curpiece_>rotate_cw()){
        throw TetrismotcursesErr("rotate_cw()");
    }
    if(!nc_.render()){
        throw TetrismotcursesErr("render()");
    }
}
```

FIGURE 84: Tetris::RotateCcw() and Tetris::RotateCw().

```

void DrawBackground(const std::string& s) { // drawn to the standard plane
    int avert;
    try{
        backg_ = std::make_unique<ncpp::Visual>(s.c_str(), &avert, 0, 0, ncpp::NCScale::Stretch);
    }catch(std::exception& e){
        throw TetrisNotcursesErr("visual(): " + s + ": " + e.what());
    }
    if(!backg_->decode(&avert)){
        throw TetrisNotcursesErr("decode(): " + s);
    }
    if(backg_->render(0, 0, -1, -1) <= 0){
        throw TetrisNotcursesErr("render(): " + s);
    }
    backg_->get_plane()->greyscale();
}

void DrawBoard() { // draw all fixed components of the game
    try{
        DrawBackground(BackgroundFile);
    }catch(TetrisNotcursesErr& e){
        stdplane_->printf(1, 1, "couldn't load %s", BackgroundFile.c_str());
    }
    int y, x;
    stdplane_->get_dim(&y, &x);
    board_top_y_ = y - (BOARD_HEIGHT + 2);
    board_ = std::make_unique<ncpp::Plane>(BOARD_HEIGHT, BOARD_WIDTH * 2, board_top_y_, x / 2 - (BOARD_WIDTH + 1));
    uint64_t channels = 0;
    channels_set_fg(&channels, 0x00b040);
    channels_set_bg_alpha(&channels, CELL_ALPHA_TRANSPARENT);
    if(!board_->double_box(0, channels, BOARD_HEIGHT - 1, BOARD_WIDTH * 2 - 1, NCBOXMASK_TOP)){
        throw TetrisNotcursesErr("double_box()");
    }
    channels_set_fg_alpha(&channels, CELL_ALPHA_TRANSPARENT);
    board_->set_base("", 0, channels);
    scoreplane_ = std::make_unique<ncpp::Plane>(2, 30, y - BOARD_HEIGHT, 2, nullptr);
    if(!scoreplane_){
        throw TetrisNotcursesErr("Plane()");
    }
    uint64_t scorechan = 0;
    channels_set_bg_alpha(&scorechan, CELL_ALPHA_TRANSPARENT);
    channels_set_fg_alpha(&scorechan, CELL_ALPHA_TRANSPARENT);
    if(!scoreplane_->set_base("", 0, scorechan)){
        throw TetrisNotcursesErr("set_base()");
    }
    scoreplane_->set_bg_alpha(CELL_ALPHA_TRANSPARENT);
    scoreplane_->set_fg(0xd040d0);
    scoreplane_->printf(0, 1, "%s", cuserid(nullptr));
    scoreplane_->set_fg(0x00d0a0);
    UpdateScore();
    if(!nc_.render()){
        throw TetrisNotcursesErr("render()");
    }
}

```

LISTING 93: Drawing the background and the gameplay plane.

```
bool InvalidMove() { // a bit wasteful, but piece are tiny
    int dy, dx;
    curpiece_>get_dim(&dy, &dx);
    while(dy--){
        int x = dx;
        while(x--){
            ncpp::Cell c, b;
            if(curpiece_>get_at(dy, x, &c) < 0){
                throw TetrisNotcursesErr("get_at()");
            }
            if(c.is_simple()){
                continue;
            }
            curpiece_>release(c);
            int transy = dy, transx = x; // need game area coordinates via translation
            curpiece_>translate(*board_, &transy, &transx);
            if(transy < 0 || transy >= board_>get_dim_y() || transx < 0 || transx >= board_>get_dim_x()){
                return true;
            }
            if(board_>get_at(transy, transx, &b) < 0){
                throw TetrisNotcursesErr("get_at()");
            }
            if(!b.is_simple()){
                return true;
            }
            board_>release(b);
        }
    }
    return false;
}
```

LISTING 94: Tetris::InvalidMove().

```

bool MoveDown() { // returns true if the game has ended as a result of this move
    int y, x;
    if(PrepForMove(&y, &x)){
        if(!curpiece_>move(y + 1, x)){
            throw TetrisNotcursesErr("move()");
        }
        if(InvalidMove()){
            if(!curpiece_>move(y, x)){
                throw TetrisNotcursesErr("move()");
            }
            if(y <= board_top_y_ - 1){
                return true;
            }
            if(LockPiece()){
                return true;
            }
            curpiece_ = NewPiece();
        }else{
            ++y;
        }
        if(!nc_.render()){
            throw TetrisNotcursesErr("render()");
        }
    }
    return false;
}

```

LISTING 95: Tetris::MoveDown().

```

void Ticker() { // FIXME ideally this would be called from constructor :/
    std::chrono::milliseconds ms;
    do{
        mtx_.lock();
        ms = msdelay_;
        mtx_.unlock();
        std::this_thread::sleep_for(ms);
        ncmtx.lock();
        if(MoveDown()){
            gameover_ = true;
            ncmtx.unlock();
            return;
        }
        ncmtx.unlock();
    }while(!gameover_);
}

```

LISTING 96: Tetris::Ticker().

```

bool LockPiece(){ // returns true if game has ended by reaching level 16
    curpiece_>mergedown(*board_);
    int bdimy, bdimx;
    board_>get_dim(&bdimy, &bdimx);
    int cleared; // how many contiguous lines were cleared
    do{
        cleared = 0;
        int y;
        StainBoard(bdimy, bdimx);
        for(y = bdimy - 2 ; y > 0 ; --y){ // get the lowest cleared area
            if(LineClear(y)){
                ++cleared;
            }else if(cleared){
                break;
            }
        }
    }
    if(cleared){ // topmost verified clear is y + 1, bottommost is y + cleared
        for(int dy = y ; dy >= 0 ; --dy){
            for(int x = 1 ; x < bdimx - 1 ; ++x){
                ncpp::Cell c;
                if(board_>get_at(dy, x, &c) < 0){
                    throw TetrisNotcursesErr("get_at()");
                }
                if(board_>putc(dy + cleared, x, &c) < 0){
                    throw TetrisNotcursesErr("putc()");
                }
                c.get().gcluster = 0;
                if(board_>putc(dy, x, &c) < 0){ // could just do this at end...
                    throw TetrisNotcursesErr("putc()");
                }
            }
        }
        linescleared_ += cleared;
        static constexpr int points[] = {50, 150, 350, 1000};
        score_ += (level_ + 1) * points[cleared - 1];
        if((level_ = linescleared_ / 10) > MAX_LEVEL){
            return true;
        }
        mtx_.lock();
        msdelay_ = std::chrono::milliseconds(Gravity(level_));
        mtx_.unlock();
        StainBoard(bdimy, bdimx);
        UpdateScore();
    }
}while(cleared);
return false;
}

```

LISTING 97: Tetris::LockPiece().

```

// tidx is an index into tetrminos. yoff and xoff are relative to the
// terminal's origin. returns colored north-facing tetrimino on a plane.
std::unique_ptr<ncpp::Plane> NewPiece() {
    // "North-facing" tetrimino forms (form in which they are released from the top) are expressed in terms of
    // two rows having between 2 and 4 columns. We map each game column to four columns and each game row to two
    // rows. Each byte of the texture maps to one 4x4 component block (and wastes 7 bits).
    static const struct tetrimino {
        unsigned color;
        const char* texture;
    } tetrminos[] = { // OITLJSZ
        { 0xc9bc900, "****"}, { 0x009caa, "  ****"}, { 0x952d98, " * ****"}, { 0xcf7900, "  ****"},
        { 0x0065bd, "*  ***"}, { 0x69be28, " **** "}, { 0xbd2939, "** * **"} };
    const int tidx = random() % 7;
    const struct tetrimino* t = &tetrminos[tidx];
    const size_t cols = strlen(t->texture);
    int y, x;
    stdplane->get_dim(&y, &x);
    const int xoff = x / 2 - BOARD_WIDTH + 2 * (random() % (BOARD_WIDTH / 2));
    std::unique_ptr<ncpp::Plane> n = std::make_unique<ncpp::Plane>(2, cols, board_top_y_ - 1, xoff, nullptr);
    if(n){
        uint64_t channels = 0;
        channels_set_bg_alpha(&channels, CELL_ALPHA_TRANSPARENT);
        channels_set_fg_alpha(&channels, CELL_ALPHA_TRANSPARENT);
        n->set_fg(t->color);
        n->set_bg_alpha(CELL_ALPHA_TRANSPARENT);
        n->set_base("", 0, channels);
        y = 0; x = 0;
        for(size_t i = 0 ; i < strlen(t->texture) ; ++i){
            if(t->texture[i] == '*'){
                if(n->putstr(y, x, "■") < 0){
                    throw TetrisNotcursesErr("putstr()");
                }
            }
            y += ((x = ((x + 2) % cols)) == 0);
        }
    }
    if(!nc_.render()){
        throw TetrisNotcursesErr("render()");
    }
    return n;
}

```

LISTING 98: Tetris::NewPiece().

```

int main(void) {
    if(setlocale(LC_ALL, "") == nullptr){
        return EXIT_FAILURE;
    }
    srand(time(nullptr));
    std::atomic_bool gameover = false;
    notcurses_options ncopts{};
    ncpp::NotCurses nc(ncopts);
    Tetris t{nc, gameover};
    std::thread tid(&Tetris::Ticker, &t);
    ncpp::Plane* stdplane = nc.get_stdplane();
    char32_t input = 0;
    ncinput ni;
    while(!gameover && (input = nc.getc(true, &ni)) != (char32_t)-1){
        if(input == 'q'){
            break;
        }
        ncmtx.lock();
        switch(input){
            case NCKEY_LEFT: case 'h': t.MoveLeft(); break;
            case NCKEY_RIGHT: case 'l': t.MoveRight(); break;
            case NCKEY_DOWN: case 'j': t.MoveDown(); break;
            case 'L': if(ni.ctrl){ notcurses_refresh(nc, nullptr, nullptr); } break;
            case 'z': t.RotateCcw(); break;
            case 'x': t.RotateCw(); break;
            default:
                stdplane->cursor_move(0, 0);
                stdplane->printf("Got unknown input U+%06x", input);
                nc.render();
                break;
        }
        ncmtx.unlock();
    }
    if(gameover || input == 'q'){ // FIXME signal it on 'q'
        gameover = true;
        tid.join();
    }else{
        return EXIT_FAILURE;
    }
    return nc.stop() ? EXIT_SUCCESS : EXIT_FAILURE;
}

```

LISTING 99: Tetris main().

Appendix A A brief history of character graphics

Let's take it back to '79!

Wu-Tang Clan ♡, “Triumph”

The earliest terminals making use of glyphs⁹⁴ printed them to paper, and are of interest to us only so far as our modern term “tty” is rather dubiously derived from “TeleTYpewriter”, as these cantankerous contraptions were known⁹⁵ (people had less experience abbreviating in those days).

These devices most typically printed 72 characters per line (CPL), a limit that has persisted in strange places[78] through the modern era. Another constant you'll see from time to time is 132 CPL, derived from line printers such as the IBM 1403, the DEC LP11, and the Centronics 101[102]. Most common, however, is the 80 column line originating in 1928's 7¾x3¼x0.007in IBM Computer Card (as designed by Clair D. Lake, borrowing from the 1890 U. S. Census cards of Herman Hollerith...themselves inspired by Joseph Jacquard's automation in 1804 of punched card loom control technology pioneered by Basile Bouchon in 1725[70]). To this day, so long as your wacky output device can do 80 columns, eh, that's good enough. In all these cases, the limit arises from the number of characters that could be printed, using the technology of the time, on their feeder paper (8.5in and 14in in the case of printers).

On, then, to the “Glass TTYs” (ugh) and Visual Display Units of the 1970s. Pictured in Figure 85 are the Computer Terminal Corporation Datapoint 3300, the Lear Sigler, Inc. ADM-3A, the Hazeltine 1500, and the Soroc IQ-120. Lacking microcontrollers, and generally implementing no independent control sequences, such devices are today often known as “dumb terminals” (this term was originally a registered trademark of Lear Sigler, see Figure 88). Already the 80x24 “standard” (it is not a standard) was emerging (the DEC contemporaries listed were already pretty “smart”, using proprietary control codes):

IBM 2260 Model 1	1965	40x6
Datapoint 3300	1969	72x25
DEC VT05	1970	72x24
IBM 3277 Model 2	1971	80x24
Texttronix 4010	1972	74x35
DECscope VT52	1974	80x24
LSI ADM-3A	1976	80x12, 80x24
Hazeltine 1500	1977	80x24
Soroc IQ-120	1977	80x24

TABLE 14: Some historical terminals and their resolutions.

Why 80x24 (or 80x25, as you'll also see)[12]? The 80 almost certainly arises from the desire to display an entire punched card (this *is* a standard—see ANSI X3.21-1967/FIPS PUB 13, “Rectangular Holes in Twelve-Row Punched Cards”)[103]. The origin of 24 is less clear. 24 is highly composite (it has more divisors than any smaller number), and it is the largest integer divisible by all natural numbers not larger than its square root. There are of course 24 hours in a day. 24 divides the scanline counts of both NTSC and PAL at 480 and 576, respectively. 24 rows of 80 columns at a byte per column utilize 93.75% of a 2KiB memory, leaving exactly 128 bytes left over, and everyone loves a good power of 2.

The Aaronites, Levite descendents of Moses's brother Aaron, the first כהן גדול (High Priest), form the priestly כֹּהֲנִים; they were divided into 24 courses. The Buddha's Dharma Chakra (Wheel of Dhamma) in its Ashoka form sends forth 24 spokes. But perhaps I grow esoteric, even speculative...in truth, 80x24 almost certainly owes its questionable existence to IBM's punched cards, IBM 2260 and 3270 wanting compatibility with IBM printers, the upstart DEC wanting compatibility with IBM software for their VT52 and legendary VT100, and the VT100 subsequently becoming a *de facto* standard for four decades.

⁹⁴Konrad Zuse's Z3, generally considered the first programmable digital computer, communicated with its operator through a matrix of blinkenlights and a not unsteampunkish keyboard that resembled the Burroughs typewriters of its era[98].

⁹⁵Though we do hear of their Snoopy calendars in the songs of legend[90].

The Dumb Terminal lets you put it all together.

With the new lower-priced Dumb Terminal™ Kit, that is. Pick one up and escape, once and for all, the headaches of scavenged teletypes and jury-rigged TV sets. With just a little time and aptitude, you can have a low and working Dumb Terminal right in your own home, garage, or business. One that lets you get it all out of your system — or into it.

Forget the cheap imitations, with their overblown price tags and interminable lists of options. With the Kit, you can build yourself the same, old basic Dumb Terminal that's been selling over 1500 units a month. With basic, sensible features like a bright 12" diagonal screen, fifty-size data entry keys, 1500 characters displayed in 24 rows of 80 letters. Plus 33 positive action switches that let you activate functions like 1 of 11 different baud rates, an RS232C interface, or a 20mA current-loop. And more. Not bad for Dumb.

All you need, besides the Kit, is some initiative, and a few basic tools — a good soldering iron, wire cutters, needle-nose pliers, and one or two trusty screwdrivers. The Dumb Terminal Kit provides you with everything else. Including an attractive cabinet, CRT screen, keyboard, PC board, and all essential electronic components. Naturally, you also get illustrated, step-by-step assembly instructions, not to mention an easy-to-understand operator's manual.

So, if you'd like more input on the Dumb Terminal Kit, just fill out the coupon and we'll send you complete, free information.

Oh, and by the way, just by sending in the coupon, you will be made a charter member of the Dumb Terminal Fan Club. A select organization that will send you your own nifty Dumb Terminal Fan Club Kit, containing an official certificate of membership, an autographed photo of the Dumb Terminal himself, and a bona fide membership card to prove irrefutably you're "One of Us." (Sorry, limit one kit per person.)

And, if you include a trifling \$6.00, you can have your very own Dumb Terminal T-shirt. (No limit at all on these.)

Simply mail the coupon and get the whole assortment. And find out why members of the Dumb Terminal Fan Club are some of the smartest people around.

ISI
Dumb Terminal.
Fun Club.

I would like more information on the Dumb Terminal Kit. And about the T-shirt. I will send you a check for \$6.00 in the Dumb Terminal Kit.

Name _____ Title _____
Address _____
City _____ State _____ Zip _____
Elected to _____ for an official Dumb Terminal T-shirt.
Please make all checks and money orders payable to L&A Inc. & Assoc.
Quantity and amount of shirt(s) required _____ M _____ XL
Check this application to: Dumb Terminal Fan Club Headquarters
c/o L&A Inc. & Assoc., P.O. Box 37123, Irvine, CA 92714

CIRCLE INQUIRY NO. 30

A Beautiful Way To Interface



IQ 140

SOROC's first and foremost concern, to design outstanding remote video displays, has resulted in the development of the IQ 140. This unit reflects exquisite appearance and performance capabilities unequalled by others on the market.

With the IQ 140, the operator is given full command over data being processed by means of a wide variety of edit, video, and mode control keys, etc.

The detachable keyboard, with its

IQ 120

The SOROC IQ 120 is the result of an industry-wide demand for a capable remote video display terminal which provides a multiple of features at a low affordable price.

We've got another new terminal and it fits right here...



It's the right terminal, with the right features, at the right price.

Announcing Hazeltine 1410.

As an industry leader, the Hazeltine 1410 terminal adds to the features available on the Hazeltine 1400 terminal. The Hazeltine 1410 terminal features a built-in printer, a built-in terminal emulator, and a built-in terminal emulator. The Hazeltine 1410 terminal is a true multi-terminal terminal. It can be used as a terminal emulator, a terminal emulator, or a terminal emulator. The Hazeltine 1410 terminal is a true multi-terminal terminal. It can be used as a terminal emulator, a terminal emulator, or a terminal emulator.

From Hazeltine — A World Leader in Information Electronics for More than a Half Century.



FIGURE 85: Clockwise starting from upper left: a dork and his LSI ADM (“American Dream Machine”, supposedly). That poor woman with the Soroc IQs looks stoned out of her gourd. Grizzly Adams rocks a Datapoint 3300, but really his mind is on seeing Skynyrd shred it this weekend. Finally, we have Frank the Cocaine Ranger and his Electric Hazeltine 1500 Band. *Gott im Himmel*, the 70s were *unseemly*.



FIGURE 86: Digital Equipment Corporation terminals of the 1970s and 1980s.



FIGURE 87: The VT220's glyphs from a ROM dump. VT100 implemented most of the first seven columns. Note the existence of box-drawing characters[75].



Vincent van Dumb.

The Dumb Terminal® video display terminal has done it again.

For around \$2000, you can have all the alphanumeric capabilities of the renowned ADM-3A Dumb Terminal, plus the full vector drawing and point plotting capabilities of a sophisticated graphics terminal. All in one neat package. That's less than half the cost of other comparably equipped graphics terminals.

The ADM-3A with Retro-Graphics™ gives you complete flexibility to develop bar charts, pie diagrams, histograms, even function plots. What's more, it's completely Tektronix® Plot 10™ software-compatible.

The package consists of an ADM-3A Dumb Terminal plus a single plug-in card engineered to fit neatly inside the ADM-3A without soldering, special tools, or a service call.

Retro-Graphics is a product of Digital Engineering, Inc., and is sold separately or installed in the ADM-3A by local Lear Siegler distributors. For the distributor nearest you, contact any Lear Siegler sales

office or Digital Engineering, Inc., 1787-K Tribute Road, Sacramento, CA 95815, 916/920-5600.

The Retro-Graphics-equipped Dumb Terminal. What does it mean to you? Draw your own conclusions.



**DUMB TERMINAL.
SMART BUY.**



LEAR SIEGLER, INC.
DATA PRODUCTS DIVISION

Lear Siegler, Inc./Data Products Division, 714 North Brookhurst Street, Anaheim, CA 92803, 800/854-3805. In California 714/774-1010. TWX: 910-591-1157. Telex: 65-5444. Regional Sales Offices: San Francisco 408/263-0506, Los Angeles 213/454-9941, Chicago 312/279-5250, Houston 713/780-2585, Philadelphia 215/245-1520, New York 212/594-6762, Boston 617/423-1510, Washington, D.C. 301/459-1826, England (04867) 80666.

DISTRIBUTORS: San Francisco, Consolidated Data Terminals, 415/533-8125. Dallas, Data Applications Corp., 214/231-4846. San Diego, Data Systems Marketing, 714/560-9222. Bedford, Continental Resources, 617/275-0850. Falls Church, Marva Data Services, 703/893-1544. Cleveland, W.C. Koepf Associates, 216/247-5129.

Dumb Terminal® is a registered trademark of Lear Siegler, Inc.

Retro-Graphics™ is a trademark of Digital Engineering, Inc.

Tektronix® and Plot 10™ are trademarks of Tektronix, Inc.

Copyrighted material

FIGURE 88: A strange time.

A.1 The DEC VTxxx terminals and ANSI X3.64-1979

Introduced in August 1978, the VT100 ushered in a new era of smart terminals using commodity (Intel) microprocessors, implementing portions of the upcoming ANSI X3.64 standard (itself based on 1976's first edition of ECMA-48) along with DEC extensions⁹⁶. This series would go on to sell over six million units, and it was a rare vendor that didn't include some degree of DEC VT compatibility. Each major iteration of the series was designed to encompass all functionality of prior iterations, beginning with the VT100's faithful emulation of the earlier era's VT52[29]. The VT102 cut down on the cost and size of the VT100, and included the 132-CPL mode by default; they were otherwise essentially the same device[30].

Terminals had quite a heyday through the 1970s and 1980s, but as the price of Wintel machines fell below \$1,000, their value proposition rapidly eroded. They lived on, especially in niche minicomputer and main-frame installations, but as of 2020 it's difficult to find a new terminal. Digital sold their terminal business to SunRiver Data Systems (now Boundless Technologies) in 1995; the boundlessterminals.com site is not offered over HTTPS, and reads "Now it is time for the last text terminals to give way to newer technologies[124]." Press F to pay respects.

The original `xterm` (released in X10R3) was written as an emulator of the VAXStation 100 (VS100), and slowly acquired scattered features from the VT100, ANSI, and other sources[38]. Thomas E. Dickey (the current maintainer of `NCURSES` and `xterm`) began working on XTerm in the mid-90s, and by 1996 had added the `decTerminalID` resource following the addition of much VT220 compatibility. `xterm` can be built with support for Sixel, ReGIS, Unicode, and an extraordinary number of archaic and/or baroque mechanics with which I'm not personally familiar. Further information is available in Mr. Dickey's [XTerm FAQ](#), which makes for excellent reading⁹⁷.

I will not attempt to list the hundreds (probably thousands) of terminal emulators that are available. Most claim some manner of "ANSI" or "vt100" (not the same thing!) compliance; take these claims with a large grain of salt. It's worth knowing about:

- `rxvt`, "Rob's `xvt`". An enhancement of `xvt` touted as a slimmed-down, simplified alternative to `xterm`. Implemented pixel-addressable graphics (in a scheme incompatible with Sixel). Like `xterm`, it uses X resources for configuration. Various forks add various essential technologies introduced since 2000.
- Konsole, the official terminal of KDE. Having never used KDE substantially, I've never made much use of Konsole. I recall it having URL recognition when most terminal emulators didn't.
- VTE terminals, a family of terminals built around GNOME's VTE library, built atop GTK3. GNOME and Xfce's terminals both wrap VTE, as do dozens of others. `xfce4-terminal` plays the role of our VTE terminal in benchmarks, because I run XFCE (atop Compiz).
- `alacritty`, written in Rust by Joe Wilm. One of the new class of emulators written directly against OpenGL.
- `kitty`, another OpenGL program, this one in C++/Python by Kovid Goyal.
- Terminology is the emulator of Enlightenment since E17, the long (twelve years!) awaited successor to E16. It seemed somewhat broken every time I tried it, and by 2014 I didn't want to spend time fixing terminal emulators. It does look really pretty on its website[91].
- Terminator was written in Java, which...life finds a way I guess.

I haven't benchmarked `rxvt` because I didn't care to figure out which of its ten thousand forks is the one you're supposed to use. Likewise, I didn't benchmark Terminator because I would prefer setting myself on fire to running a Java terminal emulator.

The best single source of terminal emulator (and terminal) information is probably the "Terminal Type Descriptions" file distributed with `NCURSES`[3].

⁹⁶The VT100 *did not* implement all of X3.64, nor was X3.64 derived from the VT100. The VT100 didn't do color, nor did it insert or delete lines. It furthermore implemented several features outside the scope of ECMA-48's first edition.

⁹⁷Mr. Dickey's XTerm and NCURSES FAQs are, in my opinion, two of the finest pieces of technical documentation in existence. It is my hope that this manuscript approaches their level.

A.2 The Curses API

Several APIs for TUIs and character (semi)graphics have emerged over the fifty years of video terminals' existence. Of them, by far the most venerable, portable, and proven is Curses, which has (for over twenty years) actually been codified in the Single UNIX Specification. Notcurses is an obvious intellectual descendant of Curses, and indeed ought to be easy to pick up by any experienced Curses programmer.

As mentioned in Chapter 6.3, Ken Arnold released the first BSD Curses library shortly after extracting `vi`'s terminal abstraction routines as `libtermcap`, and that first Curses made use of `termcap`[48]. Similarly to `termcap`, Curses had its origins in Joy's `vi` code; unlike `termcap`, Curses required significant reworking to be presented as a sensible API. Meanwhile, over in AT&T land, Mary Ann Horton (previously maintainer of BSD's `vi` and `termcap`) arrived and implemented a new Curses, this one making use of her `terminfo`. The two diverged over the years (with `PDCurses` aka Public Domain Curses reimplementing AT&T to work around licensing issues), until X/Open codified an official vendor-neutral Curses, publishing it in 1996's Issue 4, version 2 (this specification derived primarily from SVR4 Curses).

Zeyd Ben-Halim adapted `NCURSES` from the `pcurses` project of Pavel Curtis. Dickey reports the 1.8.1 version released 1993-11-05 (and included with Slackware 2.0.1) as the "first widely-used version". Numerous people (including the notorious Eric S. Raymond and Ulrich Drepper) contributed significant code, but most `NCURSES` development in recent years has been the work of Thomas Dickey.

The Curses API is best reviewed by reading the comprehensive man pages installed with `NCURSES`. A few lines follow, summarizing major differences between Curses and Notcurses:

- Curses has no built-in concept of a z-axis without use of its Panels extension, and its `stdscr` (analogous to the standard plane of Notcurses) is not considered part of the Panels stack. Curses does, however, allow windows to share memory, a capability Notcurses does not yet offer.
- Panels can be "hidden", i.e. removed from the z-axis entirely. This cannot be done in Notcurses (but planes can be hidden underneath an opaque plane, or moved outside the visible area).
- `NCURSES` defines its color API in terms of "color pairs". It is true that some terminals require both colors to be changed at the same time, but Notcurses hides this fact from the programmer, and instead allows foregrounds and backgrounds to be specified wholly independently.
- Curses does not include the Panel, Menu, or Form extensions distributed with `NCURSES`. With that said, their presence in `NCURSES` effectively means they're in Curses. Notcurses is built around a Panel-like concept, and includes menus in its base, but lacks the rich Forms capabilities of `NCURSES`.
- `NCURSES` supports only up to 256 colors at a time, as components of up to 32,767 color pairs. There are no such limits in Notcurses (assuming terminal support).
- `NCURSES` considers it an error to move a window or cursor off the screen. This is not an error in Notcurses.

Appendix B Wherein shade is thrown at terminal emulators

And as he spoke, El-ahrairah’s tail grew shining white and flashed like a star; and his back legs grew long and powerful and he thumped the hillside until the very beetles fell off the grass stems. He came out of the hole and tore across the hill faster than any creature in the world.

Richard Adams, *Watership Down*

The `notcurses-demo` program detailed in Chapter 14.1 serves as an excellent testbed for benchmarking certain properties of various terminals in various configurations. The `-c` argument ought always be provided when benchmarking, so that the PRNG is seeded with the same value⁹⁸. The `-J` argument generates JSON-formatted output suitable for machine processing. Finally, `-d` can be supplied to reduce the amount of artificial delay: the default is `-d1` for 1.0x the standard delay. `-d0` eliminates all artificial delay.

This binary is composed of over a dozen different independent demos, which can be freely scheduled on the command line. Different demos have different usefulness for benchmarking. Fixed-time, fixed-framecount demos will generally only show differences in the amount of time spent rendering. Fixed-framecount demos show difference in total time. Fixed-time demos might show differences in framecount and time spent rendering. A greater number of frames for the same demo is indicative of an advantage, as is completing any particular demo in less time. The most generally useful stats to compare across runs are average time per render and maximum time to render.

Demo	Type	Exercises
Intro	Fixed-time	Fades, EGC draws
X-Ray	Fixed-time+frame	Plane movement (x), video
Eagle	Time-to-completion	Large renders, multiple sprites
Trans	Fixed-time	Plane movement (xy)
Highcon	Time-to-completion	Intense RGB, intense redraw
Box	Time-to-completion	Intense redraw
Chunli	Fixed-time+frame	Large sprite
Grid	Time-to-completion	Intense RGB, RGB variance
Reel	Fixed-time	Plane movement (y)
Whiteout	Fixed-time	Intense font rendering
Uniblock	Fixed-time	Intense font rendering
View	Fixed-time+frame	Video
Luigi	Time-to-completion	Sprites, background
Fallin	Time-to-completion	Many planes, plane movement (y)
Sliders	Time-to-completion	Many planes
Jungle	Fixed-time	Palette cycling
Outro	Fixed-time	Fades, video

TABLE 15: Benchmarking properties of various demos. I consider the colored demos particularly informative.

Prior work benchmarking terminal emulators largely focused on “scroll speed”, a fairly useless and easily misleading statistic[7], and input latency[41].

At an 80x52 geometry, the number of bytes output by different demos spans three orders of magnitude, and the time (when all artificial delays are removed) by four orders of magnitude⁹⁹. The font family used is always Hack[94] at 10 points, except for `xterm`, which became effectively unusable with this TrueType font.

⁹⁸This only serves to eliminate variance among equal PRNG implementations.

⁹⁹With artificial delays, it’s more like a single order of magnitude.

I instead allowed `xterm` to use its default font and size, leading to a much smaller total window size at the 80x52 cell geometry. Nonetheless, `xterm` still reliably delivered the poorest performance¹⁰⁰. One interesting observation is that CPU usage of the `xorg` server process never exceeded 20% with other terminal emulators, but regularly spiked above 60% with `xterm` using bitmapped fonts, and pegged the CPU(!) with TrueType fonts (see Figure 89). This ought to be investigated.

UPDATE: I am now getting reasonable performance from `xterm` following reconfiguration of `FontConfig` to enable bitmapped fonts. The poor results seen here are not necessarily representative of a well-tempered `xterm` configuration. I am leaving them in until I can completely rerun `xterm` benchmarks, and might leave them in at that point as an alternative result set.

```
top - 08:54:49 up 1 day, 20:38, 1 user, load average: 1.24, 1.09, 0.97
Tasks: 602 total, 4 running, 597 sleeping, 0 stopped, 1 zombie
%Cpu(s): 6.1 us, 5.7 sy, 0.2 ni, 87.0 id, 0.0 wa, 0.0 hi, 1.0 si, 0.0 st
MiB Mem : 64306.3 total, 1084.7 free, 36625.3 used, 26596.2 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used, 26500.5 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
1527395 dank    20   0 311336 151264 59736 R  98.3   0.2   80:26.92 Xorg
1527776 dank    20   0 254640 168844 57572 R  75.4   0.3   61:40.76 compiz
2739006 dank    20   0  44436  17908  13900 R  57.1   0.0    0:20.30 xterm
```

FIGURE 89: Only with `xterm` do we see `xorg` and `compiz` soak cores like this. With other terminal emulators, they’re generally under 20%.

All terminals were their Debian Unstable or Arch-packaged variants at the time of testing, save `alacritty`, which was not yet present in Debian. Full details are provided in Table 16.

Emulator	Version	TERM	Comments
<code>xfce4-terminal</code>	0.8.9.1	<code>vte-256color</code>	Based on GNOME’s VTE[93]
<code>xterm</code>	Patch #353	<code>xterm-256color</code>	See note above regarding regression to bitmapped fonts, yuck.
<code>kitty</code>	0.15.0	<code>xterm-kitty</code>	OpenGL-based, C++/Python
<code>alacritty</code>	0.5.0-dev (1ddd311)	<code>alacritty</code>	OpenGL-based, Rust
<code>konsole</code>	19.08.1	<code>konsole-direct</code>	KDE’s terminal

TABLE 16: Terminal software used for benchmarking.

In order to prepare a machine for benchmarking, I disabled frequency scaling by setting all cores’ scaling governor to `performance`, and disabled boosting by writing 1 to `/sys/devices/system/cpu/intel_pstate`¹⁰¹. I disabled suspend mode on the Lenovo T580 laptop, and disabled screen blanking on both machines (I’m not sure whether this latter has any impact, but wanted to play it safe). Benchmarks always ran on the active virtual desktop. `crond` was disabled, but I did *not* disable `systemd` timers¹⁰². The runtimes were so repeatable (see e.g. Figure 92, where runs tracked so closely I thought I must have made an error) that I didn’t bother with things like CPU affinity or process priority.

It’s difficult to miss a distinct triangle pattern among the rising runtimes as width increases. In all cases where it occurs, odd widths appear to run faster than even widths. I suspect this to be a property of something within `notcurses-demo` rather than a general truth about `Notcurses`, but have not yet gotten to the root of this unexpected result. Beyond that, we also see that there is a definite difference between both

¹⁰⁰FIXME FIXME FIXME investigate this

¹⁰¹If using `acpi-cpufreq`, write 0 to `/sys/devices/system/cpu/cpufreq/boost`.

¹⁰²All the timers I had enabled are either daily or weekly, and disabling them might have cumulative performance impact vs. typical behavior. I didn’t go to any great pains to avoid running benchmarks while these tasks were running, and doubt it mattered much—they’re minor tasks, and these are 8- and 20-core machines.

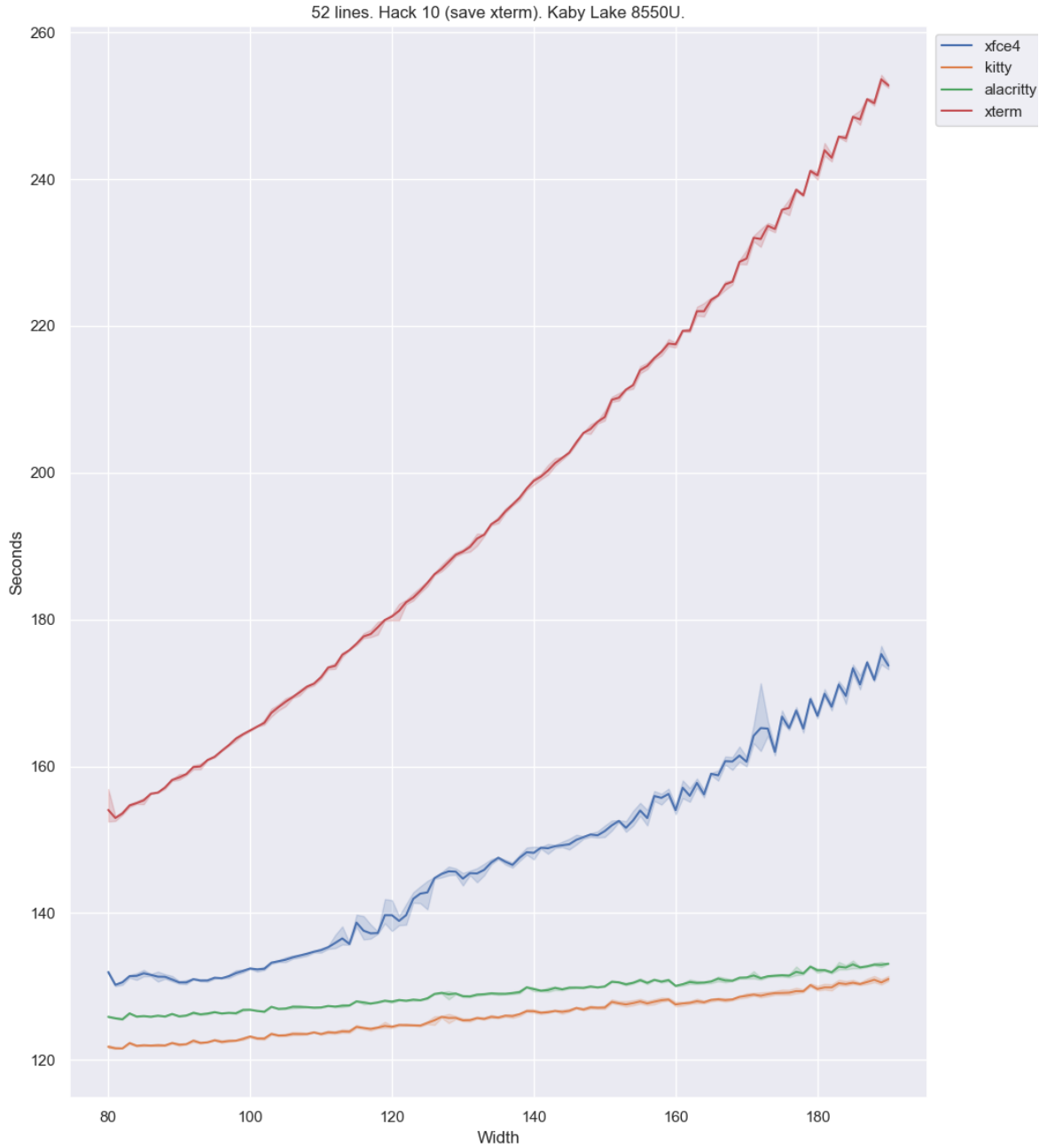


FIGURE 92: notcurses-demo runtime against terminal width. Linux 5.5.2 + i915, Arch Xorg 1.20.7, standard delays. 3 runs each, at each width. At this scale (about 100s), timings are very repeatable from run to run. Each run took about five hours.

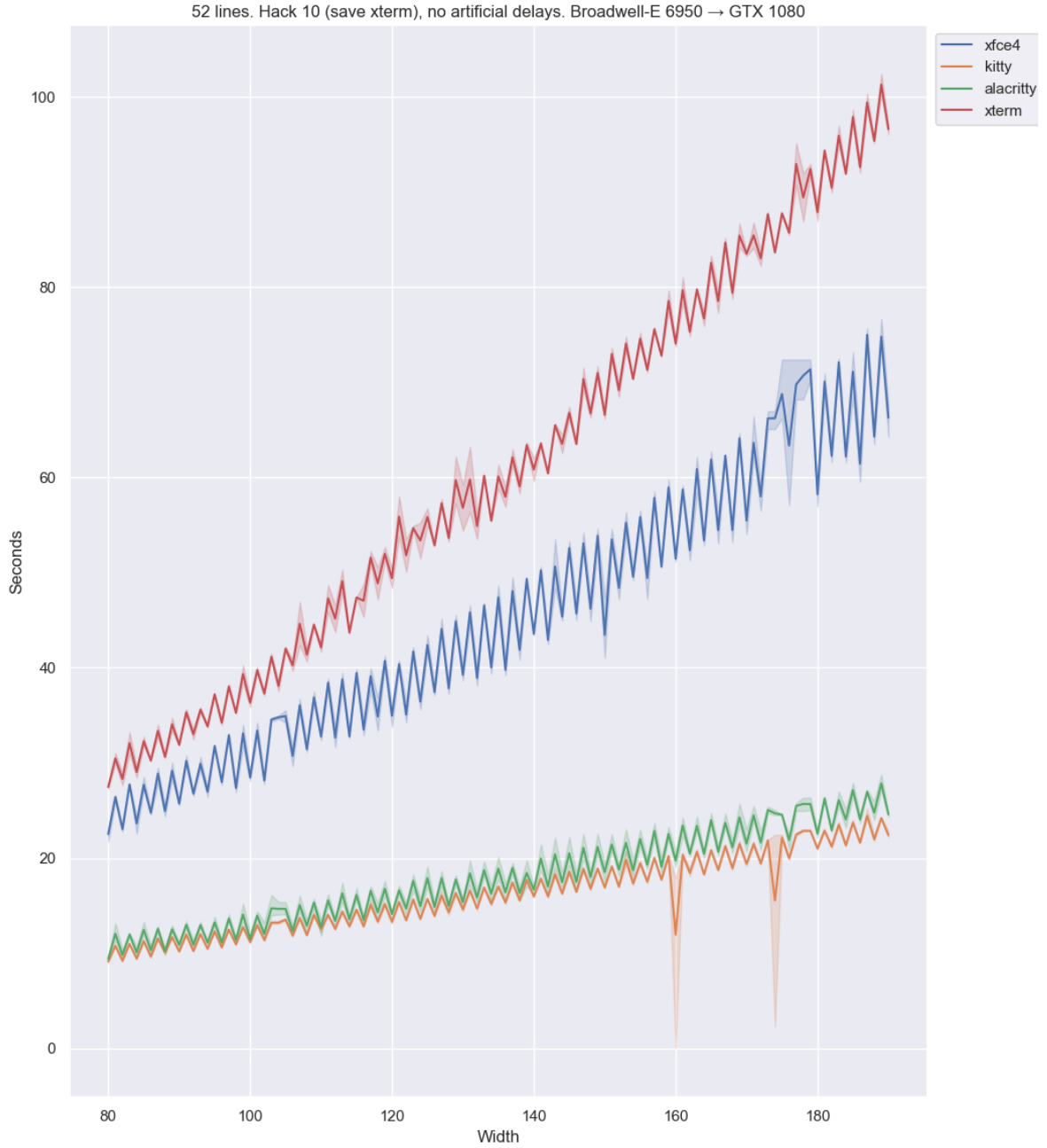


FIGURE 93: notcurses-demo runtime against terminal width. Linux 5.5.9 + NVIDIA 440.64, Debian Xorg 1.20.7, no artificial delays. 3 runs each, at each width.

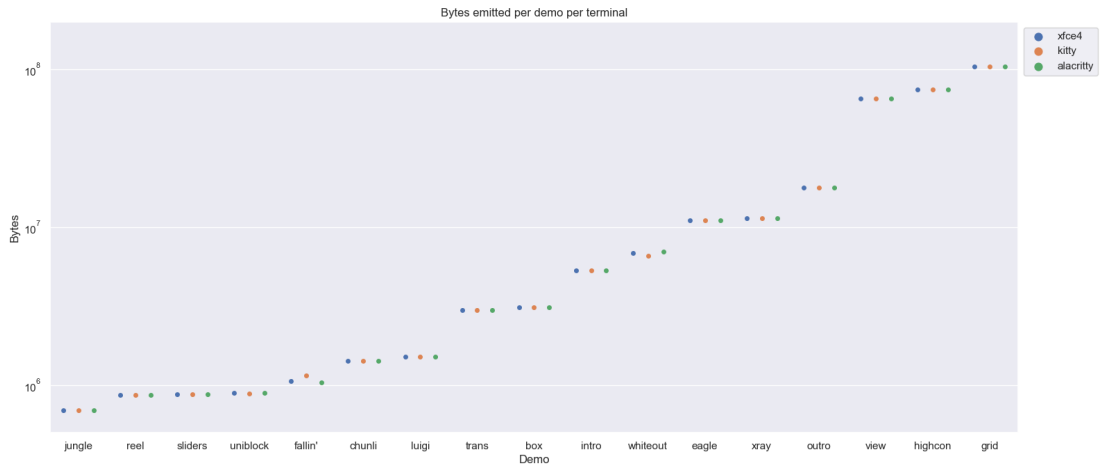


FIGURE 94: Bytes emitted on a 80x52 geometry (logarithmic y). As expected, the counts are generally equal across terminals. If plotted against width, the counts increase linearly.

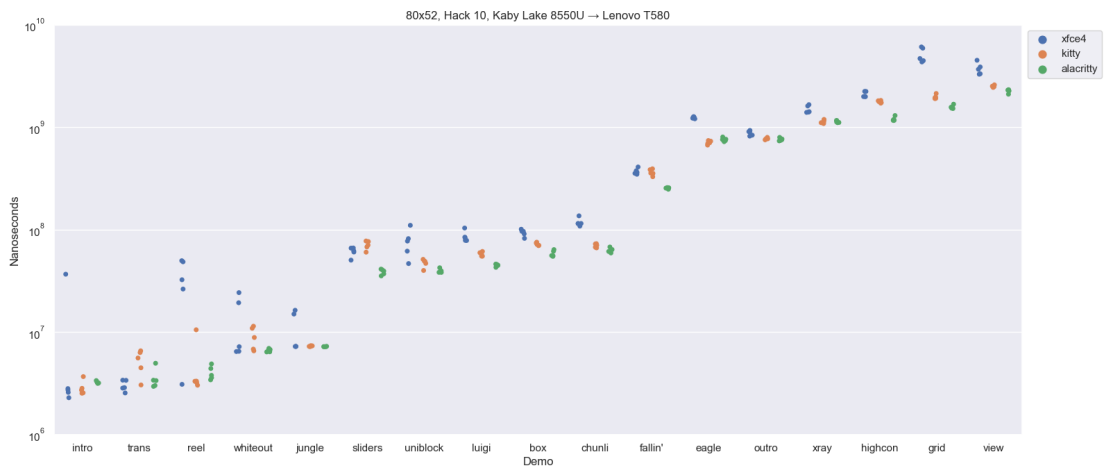


FIGURE 95: Runtimes on a 80x52 geometry (logarithmic y) atop Linux 5.5.2 + i915, Arch Xorg 1.20.7, no false delay. 5 runs each.

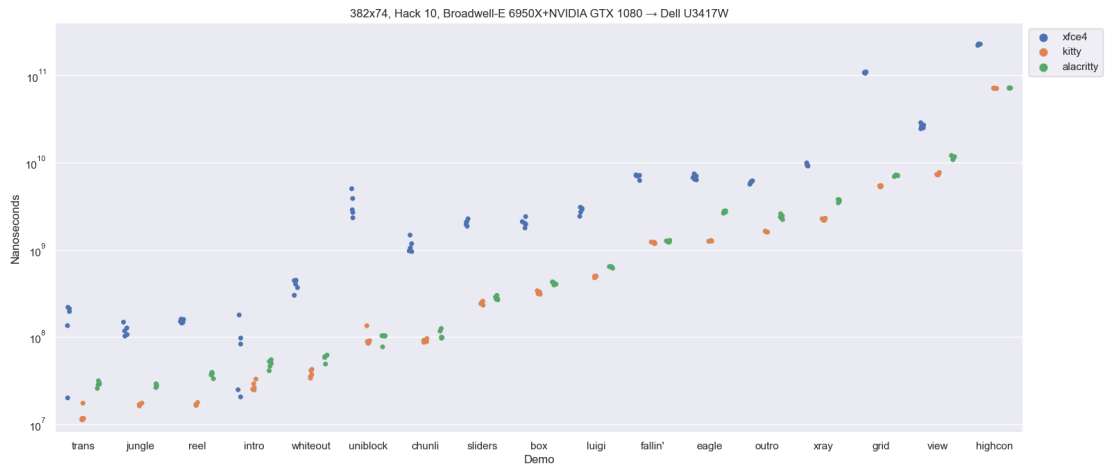


FIGURE 96: Runtimes on a 382x74 geometry (logarithmic y) atop Linux 5.5.9 + NVIDIA GTX1080 440.64, Debian Xorg 1.20.7, no false delay. 5 runs each. VTE appears to scale much more poorly than the GL-based Kitty and Alacritty.

Appendix C The Linux console

The Linux console¹⁰⁴ is substantially different from the X and Wayland terminal emulators to which one might be more accustomed¹⁰⁵. Modern terminal emulators are generally more capable than the Linux console in several ways:

- While the Linux console accepts RGB specifiers, it downsamples them to far fewer colors.
- Console font capabilities are extremely limited.

Userspace alternatives to the Linux console include `fbterm` and `kmscon`.

Like any interface to a `termios`[111] implementation, the `IUTF8` flag should be set (consult `stty`). This can be accomplished with the `IUTF8` `termios` flag (or `stty iutf8` on the command line). This is necessary for the terminal to interpret your output as multibyte UTF-8. The keyboard driver ought be placed into UTF-8 mode using the `KDSKBMODE` `ioctl`; the `kbd-mode` tool does this when invoked with `-u`. This is necessary for character erase to function properly in cooked mode. Some keyboards generate scancodes beyond the essential 128 characters, and these should be mapped to their UTF-8 equivalents. This can be accomplished with `dumpkeys` | `loadkeys --unicode`¹⁰⁶. This functionality has been supported since Linux 2.6.4, released 2004-03-11, and is almost certainly already being done in your environment.

Ensure, as always, that `LANG` is properly set, that your program initializes the locale with `setlocale(3)`, and that `TERM` is properly set (in this case, to one of the “linux*” variants).

The console font supports only 256 characters, or 512 when colors are cut in half. The built-in “PSF”¹⁰⁷ font supports the 256 characters of CP437 (remember CP437?). The font can be displayed with the `showconsolefont` command, and updated with `setfont`. The font is independent of whether the console is supplied via `vgacon` or e.g. `fbcon`. It is worth noting that this default font is missing several characters of which Notcurses makes extensive default use, particularly the rounded and double box-drawing characters. I intend to look into reprogramming the console font on the fly to better support this environment¹⁰⁸, but for the moment, Notcurses on the console is definitely a weak spot.

Consult the *The Linux Programmer’s Manual* for more information, particularly `ioctl_console(2)`[61], `ioctl_tty(2)`[62], `termios(3)`[111], `console_codes(4)`[18], and `charsets(7)`[15].

¹⁰⁴The FreeBSD console is its own bag of wonders.

¹⁰⁵Muddying the issue is the fact that video backends are sometimes described as consoles. The “Linux console” is a terminal emulator running atop some video backend—on the x86, typically either VGA Text Mode, or some trivial renderer atop a graphics-mode framebuffer (e. g. `EFIb` or `vesafb`).

¹⁰⁶If you’ve ever seen the script `unicode_start`, this is exactly what it does.

¹⁰⁷The PC Screen Font of H. Peter Anvin and Andries Brouwer.

¹⁰⁸See <https://github.com/dankamongmen/notcurses/issues/201>.

Appendix D Unicode 13

The Unicode Consortium has scheduled Unicode 13.0 for a March 2020 release. Chapters 3 and most of Chapter 4 of the Core Specification are normative. The remainder is informative. The Unicode Standard consists of the Core Specification[19], the [code charts](#), the [Unicode Standard Annexes](#), and the [Unicode Character Database \(UCD\)](#).

A Unicode Standard Annex (UAX) forms an integral part of the Unicode Standard, but is published online as a separate document. The Unicode Standard may require conformance to normative content in a Unicode Standard Annex, if so specified in the Conformance chapter of that version of the Unicode Standard. The version number of a UAX document corresponds to the version of the Unicode Standard of which it forms a part.

UAX #9	Unicode Bidirectional Algorithm
UTS #10	Unicode Collation Algorithm
UAX #11	East Asian Width
UAX #14	Unicode Line Breaking Algorithm
UAX #15	Unicode Normalization Forms
UAX #24	Unicode Script Property
UAX #29	Unicode Text Segmentation
UAX #31	Unicode Identifier and Pattern Syntax
UAX #34	Unicode Named Character Sequences
UAX #38	Unicode Han Database (UniHan)
UTS #39	Unicode Security Mechanisms
UAX #41	Common References for Unicode Standard Annexes
UAX #42	Unicode Character Database in XML
UAX #44	Unicode Character Database
UAX #45	U-Source Ideographs
UTS #46	Unicode IDNA Compatibility Processing
UAX #50	Unicode Vertical Text Layout
UTS #51	Unicode Emoji

TABLE 17: Unicode 13.0.0 Standard Annexes and Synchronized Technical Standards.

Appendix E Relevant Standards

All major standards regarding character sets, control sequences, and their encoding have ISO/IEC versions, which should probably be considered their canonical editions. Many of these ISO/IEC standards ratified or included ECMA or ANSI standards. Table 18 lists equivalencies.

ISO/IEC	ANSI	ECMA	Topic
646:1991	X3.4-1986	6:1991	7-bit character sets Twinned with ITU T.50
2022:1994	x	35:1994	8-bit character sets
2375:2003	x	x	Registration of character sets
4873:1991	x	43:1991	Multitiered 8-bit codes
6429:1992	X3.64-1979	48:1991	Control codes
8613-6:1994	x	x	Graphic renditions in terms of SGR Twinned to ITU T.416
8859:1998	x	94:1986	Official 8-bit character sets ECMA 96 is only №1 to №4.
10367:1991	x	x	G0, G1, G2, and G3
10646:2017	x	x	The Universal Character Set
14755:1996	x	x	Input methods for the UCS

TABLE 18: ECMA/ANSI/ISO standards referenced in this text.

POSIX is a family of standards from IEEE 1003, first released as IEEE 1003.1-1988 as:

- POSIX.1/IEEE 1003.1-1988: Core Services, incorporating ANSI C 1989
- POSIX.2/IEEE 1003.2-1992: Shell and utilities
- POSIX.1b/IEEE 1003.1b-1993: Real-time extensions
- POSIX.1c/IEEE 1003.1c-1995: Threads

Starting with the 1997’s SUS2 (“UNIX 98”), it was agreed that the Austin Group would take over development of POSIX. Releases of the standard would be done under the SUS aegis, and then made POSIX standards upon ISO ratification. Three releases have followed—POSIX.1-2001 and its amending 2004 Technical Corrigendae, POSIX.1-2008 (aka “Base Specifications Issue 7”) and *its* two amendments, and POSIX.1-2017.

The Single Unix Specification, then, is developed and issued by the Austin Group, a combination of ISO JTC 1 SC22, the Open Group, and ISO 1003. The first SUS emerged in 1995 as a ratification of the Open Group’s Portability Guide (XPG) 4 version 2. SUS2 followed in 1997, as mentioned, and in 2001 came the glorious unification: SUS3 aka POSIX.1-2001. SUS4 was built atop POSIX.1-2008, and has seen three new editions in 2013, 2016, and 2018. If you comply with SUS4 including both its Technical Corrigendae, you’re officially “UNIX V7”, and are out a few tens of thousands of dollars in Open Group conformance testing.

Glossary of terms

When possible, I have followed the definitions of RFC 2978[45] and the Glossary of Unicode Terms[22].

alternate screen A capability of some terminals (described by the `smcup` termcap capability and `enter_ca_mode` terminfo capability). Entering the alternate screen sees the screen cleared, and output will not be added to the scrollback buffer. Output to the alternate screen is no longer visible upon leaving it. Whether or not the original screen contents are restored is terminal dependent; if the `non_rev_rmcup` terminfo capability is defined, the original screen contents will *not* be restored.

ANSI X3.4-1986 “Coded character set—7-bit American National Standard Code for Information Interchange”. A 128-element codeset and a 7-bit character encoding perhaps better known as ASCII[59]. Its code set is a proper subset of UCS, and its encoding is a proper subset of UTF-8 (but *not* UTF-16, nor UTF-32). In terms of [standard character sets](#), ASCII is the union of ISO/IEC 646:1991’s dotriacontacharacter C0 control set (ISO registration 001) and a 94-character graphics set (ISO registration 006).

ANSI X3.64-1979 “Control Sequences for Video Terminals and Peripherals”. See ECMA-48:1991, which it ratified.

ASCII See [ANSI X3.4-1986](#).

bounding box The smallest rectilinear area necessary for containing some sprite or other displayed object.

C0 The first 32 values (hex 0x00–0x1f) of the 7-bit ISO/IEC 646:1991, including perennial favorites NUL, BEL, BS, HT, LF, VT, FF, and CR. ISO registration number 001.

canonical mode See [cooked mode](#).

cbreak mode A kind of middleground between [cooked mode](#) and [raw mode](#), cbreak mode disables line buffering and erase/kill character processing. Interrupt and flow control characters are unaffected. Primarily identified with (and defined in) Curses[109], and sometimes known as “rare mode”.

CDK The Curses Development Kit, a widget library developed against the Curses specification. It is not distributed as part of NCURSES, though both are maintained by Thomas E. Dickey.

cell The rectilinear area corresponding to a given text-mode coordinate. The analogue of pixels in graphics mode, a cell typically contains more pixels along its height than its width. 12 lines of 6 pixels each is not unreasonable. A cell can usually represent a grapheme cluster, a foreground color, a background color, and some number of attributes, independently from surrounding cells.

character One of the most imprecise terms in computer science (see Chapter 7). The C and C++ languages have a data type `char`, which usually cannot hold the world’s characters. Whether it is signed or unsigned is implementation-defined. Should I speak of them, I’ll write `char`. When I use the word “character”, I mean either an element of some code set, or such an element as encoded by some character encoding. The context ought make obvious which meaning is intended.

character encoding A mapping from one or more [code sets](#) to a set of octet sequences. Also known as a **character encoding scheme**[45].

CIELAB A 1976 color space (also known as “CIE L*a*b” or “Lab”) defined such that numerical changes correspond to roughly the same perceptible change. It is a more perceptually uniform derivative of 1931’s CIEXYZ, influenced by the Munsell color system. CIELAB colors are not absolute, but defined relative to a CIEXYZ white point. The International Commission on Illumination further developed CIEXYZ and CIELAB into the 2002 CIECAM02 color appearance model.

- code point** A numerical value within a code space. In and of itself, it has neither an associated sequence of bits nor any particular glyph. Indeed, a code point might correspond to some non-printable control.
- code set** A set of abstract characters, each having a name and a code point. [ISO/IEC 10646:2017](#) specifies almost 150,000 abstract characters as the Universal Coded Character Set (UCS). This is smaller than its corresponding code space by about an order of magnitude. Also known as a **coded character set**[\[45\]](#).
- code space** A numerical range spanning all the code points of a code set. The Unicode code space (as of Unicode 13) is made up of 17 contiguous planes of 65,536 contiguous code points each, for a size of $17 * 2^{16}$.
- console** Though sometimes used synonymously with “terminal”, console almost always refers to text-based environments on personal computers. On Linux x86 machines, this could mean a serial terminal, an AT keyboard and a VGA in 80x25 text mode, a USB keyboard with a UEFI framebuffer, or any number of other things. When I use “console”, I mean terminals outside of a GUI context.
- cooked mode** The default terminal mode under SUSv4 (compare [raw mode](#) and [cbreak mode](#)). The terminal driver buffers input until a newline is entered, while echoing it to the screen. Ctrl+C is mapped to SIGINT, Ctrl+\ is mapped to SIGQUIT, and Ctrl+Z is mapped to SIGTSTP. Buffered input is flushed when these signals are sent. Also known as [canonical mode](#).
- Curses** The X/OSI specification of an API for optimized cursor routines. It was most recently ratified as part of the January 2018 Single UNIX Specification version 4 administrative rollup, aka “susv4-2018”. NCURSES is an implementation of Curses (plus extensions). Notcurses is not.
- cursor** Both the location where output will next be written, and possibly a visual indication of that location.
- dialog** An NCURSES-based application for TUI modal dialogs.
- DirectColor** In an Xorg context, DirectColor is a strange color system (“visual type”) of Xlib[\[1\]](#) involving tripartite palette indexing. In an [ISO/IEC 8613-6:1994](#) context, 24-bit RGB, which in an Xorg context is “TrueColor”. I eschew “DirectColor”, and stick instead with [TrueColor](#).
- EBCDIC** IBM’s Extended Binary Coded Decimal Interchange Codes, announced alongside the legendary System/360, extended the BCDICs into a collection of 8-bit codes. Not a single graphic code matches its ASCII equivalent, in part because EBCDIC has properties including a discontinuous Latin alphabet. EBCDIC is thankfully rare in modern practice, living on primarily as a boogiemane with which the owners of standard library factories frighten students learning C. Exhaustive information regarding EBCDIC can be found in [\[79\]](#).
- ECMA-35:1994** “Character Code Structure and Extension Techniques”, 6th edition. In the words of Mike Frysinger, it “covers some fundamentals related to character sets, the notions of C0, C1, G0 (but not their contents), and how escape sequences work (but not what they mean)[\[46\]](#).” Ratified [ISO/IEC 2022:1994](#).
- ECMA-43:1991** “8-bit Coded Character Set Structure and Rules”, 3rd edition. Extends [ECMA-6:1991](#), conforms to [ECMA-35:1994](#). Also published as [ISO/IEC 4873:1991](#).
- ECMA-48:1991** “Control Functions for Coded Character Sets”, 5th edition. Control functions and their coded representations for use with ECMA-35-compliant 7-bit and 8-bit codes. Ratified as [ISO/IEC 6429:1992](#).
- ECMA-6:1991** “7-bit Coded Character Set”, 6th edition. Ratified [ISO/IEC 646:1991](#).
- ECMA-94:1986** “8-bit Single Byte Coded Graphic Character Sets—Latin Alphabets № 1 to № 4.”, 2nd edition. Ratified [ISO/IEC 8859:1998-1](#), -2, -3, and -4.
- em** A typographic term referring to the width of a glyph. It is equal to the current point size. It derives its name from the width of a capital M, which was often the widest glyph in a typeface.

- extended grapheme cluster** A “user-perceived character” in the words of Unicode Standard Annex #29[25]. For purposes of Notcurses, an EGC atomically inhabits a cell. It cannot be partially overwritten, and emitting one always advances the cursor.
- gamma correction** Compression and decompression based around the nonlinear nature of human color perception. The eye has more sensitivity to differences between darker shades than lighter ones.
- glyph** A visualization of an element of a language..
- HSV** The Hue, Saturation, and Value color model. A transformation of **RGB**.
- ISO/IEC 10367:1991** “Information technology—Standardized coded graphic character sets for use in 8-bit codes”. Realizes Layers 2 and 3 of the [ISO/IEC 4873:1991](#) specification by defining G0, G1, G2, and G3 sets.
- ISO/IEC 10646:2017** “Information technology—Universal Coded Character Set (UCS)”. The **code set** forming the backbone of Unicode. It is almost always best encoded as UTF-8.
- ISO/IEC 14755:1996** “Information technology—Input methods to enter characters from the repertoire of ISO/IEC 10646 with a keyboard or other input devices”.
- ISO/IEC 2022:1994** “Information technology—Character code structure and extension techniques”. An absolute mess of a standard. The bridge between [ISO/IEC 646:1991](#) and [ISO/IEC 8859:1998](#). Ratified as [ECMA-35:1994](#).
- ISO/IEC 2375:2003** “Information technology—Procedure for registration of escape sequences and coded character sets”. Policies governing registration for [ISO/IEC 646:1991](#) and [ISO/IEC 2022:1994](#).
- ISO/IEC 4873:1991** “Information technology—ISO 8-bit code for information interchange—Structure and rules for implementation”. Extends [ISO/IEC 646:1991](#). Defines three “levels” of 8-bit encodings, which you can read about if you want to (I don’t recommend it). Level 1 is realized by [ISO/IEC 8859:1998](#). Levels 2 and 3 are realized by [ISO/IEC 10367:1991](#). Also published as [ECMA-43:1991](#).
- ISO/IEC 6429:1992** “Information technology—Control functions for coded character sets”. Ratified [ECMA-48:1991](#).
- ISO/IEC 646:1991** “Information technology—ISO 7-bit coded character set for information interchange”. A seven-bit encoding developed in conjunction with ASCII. Of its 128 codes, 116 are invariant characters, and the remaining 12 are defined by national ISO 646 variants. All variants have 95 graphic characters (space and 94 printables), and 33 control characters (these definitions give rise to ANSI C’s `ctype` definitions). The only full 128-character encoding defined in the document is the International Reference Version; as of the most recent (1991) edition, the IRV is explicitly identical to ASCII. [ISO/IEC 646:1991](#) is twinned with [ITU T.50](#), and ratified as [ECMA-6:1991](#).
- ISO/IEC 8613-6:1994** “Information technology—Open Document Architecture (ODA) and Interchange Format: Character content architectures”. Defines many graphic renditions in terms of **SGR**. Referenced by [ECMA-48:1991](#) in terms of future standardization (for **SGR** attributes 38 and 48, sometimes mistakenly called “ANSI color”). Twinned to [ITU T.416](#).
- ISO/IEC 8859:1998** “Information technology—8-bit single-byte coded graphic character sets”. A set of 15 standards, emerging from 1987 to 2001, of which [ISO 8859-1:1987](#) (Part 1: Latin alphabet № 1) is probably the best-known. 8859 covers only graphic characters, not control characters. In 1992, [ISO-8859-1](#) (note the extra hyphen compared to [ISO 8859-1](#); egads), was registered, mating [ISO 8859-1](#) with [ISO/IEC 646:1991](#)’s **C0** and [ISO/IEC 6429:1992](#)’s **C1** for a complete 256-code encoding. This result is a proper subset of UTF-8, and a proper superset of ASCII. The first four were ratified as [ECMA-94:1986](#).

ISO-8859-1 IANA preferred name for “Latin alphabet № 1” from ISO/IEC 8859-1, with C0 and C1 from ISO/IEC 6429:1992. Quoth Frank da Cruz: “Similarly, ISO 8859-1 Latin Alphabet 1, which most of us think of as a coherent 8-bit character set is, in truth, composed of the ISO/IEC 646:1991 control set (registration 001), the ISO 646 USA graphics set (006), the characters Space and Delete, a second 32-character control set (normally, but not necessarily, ISO/IEC 6429:1992, registration 077), and a 96-character set known as “The Right-hand Part of Latin Alphabet 1” (registration 100). Each of these pieces except Space and Delete has its own unique registration number and designating escape sequence. There is no single, unique identifier for the 8-bit Latin-1 character set in its entirety[33].” .

ITU T.416 Twinned to ISO/IEC 8613-6:1994.

ITU T.50 The International Reference Alphabet (IRA), formerly known as International Alphabet № 5. Its most recent (1992) edition is twinned to ISO/IEC 646:1991:1991, and thus (despite its name) leaves 12 positions open for national variants.

keyboard An input device primarily used to select from a set of characters. These characters might or might not correspond to elements in the active encoding. Unlike pointers, keyboards do not indicate position.

NCURSES An implementation of X/OSI Curses (plus extensions) maintained by Thomas E. Dickey under the auspices of the Free Software Foundation. Installed on just about every currently-running UNIX system, it is a venerable, proven library, and the foundation of a great many applications.

newt A minimal TUI library written atop slang, providing the foundation for `whiptail`. It is not a Curses implementation.

non-canonical mode See [raw mode](#). This is an ambiguous term, since neither [cbreak mode](#) nor [raw mode](#) is canonical—eschew it.

Notcurses The baddest character graphics/TUI library in town.

pointer An input device primarily used to indicate position. It corresponds with one cell at any given time, which might or might not be independent of the cursor. There can be more than one pointer. Like a [keyboard](#), pointers typically have one or more distinct buttons—though usually far fewer than a keyboard.

process group A set of processes sharing the same PGID. When the terminal generates a signal for a process (e.g. `SIGINT` in response to `Ctrl+C`), it sends the signal to all processes in that process group. Of the process groups making up a session, only one is the foreground process group. Attempts to perform terminal I/O by session processes outside the foreground process group will result in `SIGTTIN` or `SIGTTOU`.

pseudoterminal “A pair of virtual character devices that provide a bidirectional communication channel. One end of the channel is called the master; the other end is called the slave. The slave end of the pseudoterminal provides an interface that behaves exactly like a classical terminal. A process that expects to be connected to a terminal, can open the slave end of a pseudoterminal and then be driven by a program that has opened the master end. Anything that is written on the master end is provided to the process on the slave end as though it was input typed on a terminal.” (`pty(7)`, *Linux Programmer’s Manual*[96])

Pseudoterminals are used by terminal emulators and interactive SSH connections. They are essentially buffers with some basic transformation capabilities. `Ctrl-C` is turned into `SIGINT` by the pseudoterminal device, and it handles relevant `ioctl()`s.

raw mode A collection of behaviors differing from [cooked mode](#) or [cbreak mode](#), originating in the Version 7 terminal driver. Notably, the terminal driver no longer buffers input or generates signals. Raw mode can be entered with the `cfmakeraw()` termios call, where available.

RGB The Red, Green, Blue color model. In the context of Notcurses, this almost always means a 24-bit value, with 8 bits per component (some contexts accept a 32-bit value, where the least significant 8 bits are ignored).

SGR Select Graphic Rendition. An ANSI escape (defined in [ECMA-48:1991](#)) and implemented (to one degree or another) by the VT100[30] (and most terminals since)¹⁰⁹. 64 different attributes are defined by ECMA-48, including e. g. underline and eight colors (plus a default color). Sometimes the “bold” (1) and “faint” (2) attributes of SGR cause a different color to be selected; sometimes they change the glyph.

standard character sets Any character set registered in the *ISO Register of Coded Characters to be Used With Escape Sequences* (ISO-IR) via the provisions of [ISO/IEC 2375:2003](#). ISO standard character sets are either “control sets” or “graphic sets” per [ISO/IEC 2022:1994](#).

termcap A deprecated terminal capability database, superseded by [terminfo](#).

terminal In the abstract, a means for entering data into a computer (typically involving a keyboard, and often a pointing device), and a means for displaying that computer’s output. In the concrete, electromechanical devices for doing the same. I use “terminal” in the abstract sense throughout this book, but both meanings are in common use:

- I pull my chair up to my desk and *engage my terminal*.
- I press Meta+F1 and *bring up a terminal*.

Only the mad might refer to a physical device as a terminal emulator.

terminal emulator A program in a GUI environment providing a two-dimensional array of cells. A terminal emulator differs from a framebuffer/canvas in that each of these cells contains rendered (and possibly stylized) font glyphs, as opposed to a single pixel. Terminal emulator processes are connected to the slave ends of pseudoterminals.

The pure character graphics environment of e.g. Linux or FreeBSD outside of X11 or Wayland is technically a terminal emulator, but almost always referred to as a “console”. When I say “terminal emulator” in this book, I do not mean to include consoles.

terminfo A database describing terminals and their capabilities. It is primarily indexed by terminal name, yielding an entry covering several hundred capabilities. These capabilities are then indexed by name. A terminal environment is responsible for setting the `TERM` environment variable to the correct primary terminfo index. `termcap` has been deprecated by the introduction of `terminfo`, which is superior in every way.

termios A POSIX API and data structure for working with arbitrary terminals. These functions all start with `tc` or `cf`, and control low-level behavior. See `termios(3)`[111].

TrueColor Specification of RGB with 24 bits in 3 8-bit channels, or RGBA with 32 bits in 4 8-bit channels[104]. Notcurses supports direct 24-bit TrueColor specification and 2 bits of alpha. NCURSES (in its extended mode) supports 24-bit TrueColor specification of palettes of up to 256 colors expressed as up to 32,768 color pairs.

UCS See [ISO/IEC 10646:2017](#).

UCS-2 A fixed-length encoding of the Basic Multilingual Plane. Each character is encoded to a single 16-bit code unit, for a total of 2 bytes per encoded character. UCS-2 cannot encode all of UCS, and is infrequently used. An amendment to [ISO/IEC 10646:2017](#) extended UCS-2 to [UTF-16](#).

UCS-4 A fixed-length encoding of [UCS](#) to 32-bit code units. [UTF-32](#) is a proper subset of UCS-4[122].

¹⁰⁹Support is quite varied among terminals. VT100 didn’t even do color.

Unicode UCS as defined by [ISO/IEC 10646:2017](#) (which Unicode inducts), plus rules on how to use that code set. Unicode is defined by the Unicode Consortium.

UTF-1 An early garbage form of UTF-8. [ISO/IEC 10646:2017](#) Annex G, since deprecated. Unfortunately, the page might not yet have been ripped out at your local library. Don't use UTF-1.

UTF-16 A variable-length encoding of UCS. Each character is encoded to either one or two 16-bit code units, for a total of 2 or 4 bytes per encoded character. For fun, it comes in two endiannesses. There's very little good about UTF-16, and it ought neither be purchased, nor accepted as a gift. Perhaps not coincidentally, it is the internal encoding of both Microsoft Windows and Java.

UTF-32 A fixed-length encoding of UCS. Each character is encoded to a single 32-bit code unit, for a total of 4 bytes per encoded character. UTF-32 is a restricted subset of [UCS-4](#)^[122].

UTF-8 “The problems outlined here go away when exclusively using UTF-8, which is one of the many reasons that it is now the mandatory encoding for all things.”—WHATWG Encoding Standard^[73]. Really, its the only Unicode encoding in which you ever ought pass information between tools^[97].

virtual terminal Some consoles multiplex multiple instances. Each is typically referred to as a virtual terminal. I have never heard anyone refer to the multiple tabs of a terminal emulator as “virtual terminals”, and would look at them strangely if I did.

whiptail An application for providing TUI modal dialogs, similar to `dialog`. It is written using Newt.

wide character This has at least three distinct meanings:

- `wchar_t`: A data type in C (where it is an alias of an integral type) and C++ (where it is a fundamental type). `wchar_t` by the Standard ought be “large enough to represent any supported character code point”. The Redmond brain trust turned around and helpfully defined `wchar_t` to be 16 bits on the UTF-16-blighted Microsoft Windows, where it cannot represent code points beyond the Basic Multilingual Plane. Use `char32_t` if you need a type that can represent any Unicode point. Unfortunately, standard C interfaces such as `wcwidth(3)` want `wchar_t`. A “wide character” in the C/C++ context means “a character from a set which cannot be represented using the `char` type.” Nothing is implied about the column width or the encoded byte length of a given character.
- **Wide East Asian characters**: Defined in Unicode Standard Annex #11. Every Unicode character has as its *default width* property one of six values: *Ambiguous*, *Fullwidth* (全形/全角), *Halfwidth* (半形/半角), *Narrow*, *Wide*, and *Neutral* aka “Not East Asian”. All six resolve to either “narrow” or “wide”, with the resolution depending on context. All fullwidth characters are wide, and all halfwidth characters are narrow. Traditionally, “narrow” characters occupy half an `em` in East Asian writing, but today it is more common for “wide” characters to occupy two `ems`.
- **Multicolumn EGCs**: Depending on the font, an EGC might occupy more than one column. The ANSI/ISO C function `wcwidth(3)` supposedly returns the number of columns a `wchar_t` will occupy, but it can't be trusted—this is ultimately a property of the font. Perhaps most spectacular is U+FD0D ARABIC LIGATURE BISMILLAH AR-RAHMAN AR-RAHEEM aka

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

References

- [1] “3.1 Visual Types.” In: *The Xlib Manual*. URL: <https://tronche.com/gui/x/xlib/window/visual-types.html>.
- [2] Linus Akesson. “The TTY demystified (Accessed 2020-02-26).” In: (). URL: <http://www.linusakesson.net/programming/tty/index.php>.
- [3] Thomas E. Dickey et al. “TERMINAL TYPE DESCRIPTIONS SOURCE FILE.” In: (2019-09). URL: <https://invisible-island.net/ncurses/terminfo.src.html>.
- [4] Kenneth C. R. C. Arnold. *Screen Updating and Cursor Movement Optimization: A Library Package*. Tech. rep. 1977.
- [5] European Computer Manufacturers Association. *ECMA Standard 48: control functions for coded character sets*. Tech. rep. 1991. URL: <http://www.ecma-international.org/publications/files/ecma-st/Ecma-048.pdf>.
- [6] Video Electronics Standards Association. “Super VGA BIOS Extension Standard.” In: VS891001 (1989-10).
- [7] Antoine Beaupré. *A look at terminal emulators, part 2*. Tech. rep. 2018-05. URL: <https://anarc.at/blog/2018-05-04-terminal-emulators-2/>.
- [8] Joseph D. Becker. *Unicode 88*. 1988-08.
- [9] Robert Bemer. “A proposal for character code compatability.” In: (1960-02). URL: <https://dl.acm.org/doi/10.1145/366959.366961>.
- [10] Nick Black. *New Directions in Window Management, Part I*. Tech. rep. 2013-03. URL: <https://www.sprezzatech.com/blog/0014-new-directions-in-window-management-p1.html>.
- [11] Nick Black and Richard Vuduc. *libtorque: Portable Multithreaded Continuations for Scalable Event-Driven Programs*. Tech. rep. 2010. URL: <https://nick-black.com/dankwiki/images/b/be/Hotpar2010.pdf>.
- [12] Ray Borrill. “Letters to the Editor: 80 Column Controversy, Letter 1.” In: *InfoWorld* 3.38 (1981-11), pp. 44–45.
- [13] Jonathan de Boyne Pollard. “TERM, COLORTERM, XTERM_VERSION, VTE_VERSION—terminal type environment variables.” In: *nosh Guide*.
- [14] Frederick P. Brooks. *The Mythical Man-Month (Anniversary Ed.)* USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201835959.
- [15] “charsets(7)—character set standards and internationalization.” In: *Linux Programmer’s Manual*.
- [16] “clock_gettime(2)—clock and time functions.” In: *Linux Programmer’s Manual*.
- [17] “clone(2)—create a child process.” In: *Linux Programmer’s Manual*.
- [18] “console_codes(4)—Linux console escape and control sequences.” In: *Linux Programmer’s Manual*.
- [19] The Unicode Consortium. *The Unicode Standard, Version 13.0.0*. 2020-03. URL: <https://unicode.org/versions/Unicode13.0.0/>.
- [20] Unicode Consortium. *Early Years of Unicode (Accessed 2020-03-09)*. Tech. rep. URL: <http://www.unicode.org/history/earlyyears.html>.
- [21] Unicode Consortium. *Emoji Versions & Sources*. Version 13.0.0. 2020. URL: <https://www.unicode.org/emoji/charts-13.0/emoji-versions-sources.html>.
- [22] Unicode Consortium. *Glossary of Unicode Terms (Accessed 2020-02-15)*. URL: <http://www.unicode.org/glossary/>.
- [23] Unicode Consortium. *The Unicode Standard—Core Specification*. Version 13.0.0. 2020. URL: <https://unicode.org/versions/Unicode13.0.0>.
- [24] Unicode Consortium. *Unicode Standard Annex #15: Unicode Normalization Forms*. Version 12.0.0. 2019. URL: <https://unicode.org/reports/tr15/>.
- [25] Unicode Consortium. *Unicode Standard Annex #29: Unicode Text Segmentation*. Version 12.0.0. 2019. URL: <https://unicode.org/reports/tr29/>.
- [26] Unicode Consortium. *Unicode Standard Annex #51: Unicode Emoji*. Version 13.0.0. 2020-02. URL: <https://unicode.org/reports/tr51/>.
- [27] Alan Coopersmith. *The X New Developer’s Guide: X Window System Concepts*. Tech. rep. 2013. URL: <https://www.x.org/wiki/guide/concepts/>.
- [28] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd ed. O’Reilly Media, 2005-02. ISBN: 978-0596005900.
- [29] Digital Equipment Corporation. *DECscope User’s Manual*. EK-VT5X-OP-001. 1976.

- [30] Digital Equipment Corporation. *VT100 User Guide*. 1979.
- [31] IBM Corporation. *Graphic Character Sets and Code Pages, Personal Computer*. Tech. rep. 00437. 1986. URL: <ftp://ftp.software.ibm.com/software/globalization/gcoc/attachments/CP00437.txt>.
- [32] Mark Crispin. “Re: TECO and mung command.” In: *alt.sys.pdp10* 1155 (1995-09). URL: <http://www.inwap.com/pdp10/usenet/its>.
- [33] Frank da Cruz. “Interchange of Non-English Computer Text.” In: (1994). URL: <http://www.columbia.edu/kermit/accents.html>.
- [34] M.Z. Danielewski. *House of Leaves*. Doubleday, 2000. ISBN: 9780385603102.
- [35] *Debian Social Contract*. Version 1.1. 2004-04. URL: https://www.debian.org/social_contract.
- [36] Jonathan Dent. *No tears of joy (yet): emoji make their OED debut*. Tech. rep. 2019-06. URL: <https://public.oed.com/blog/no-tears-of-joy-yet-emoji-make-their-oed-debut/>.
- [37] Thomas E. Dickey. *NCURSES—Frequently Asked Questions*. URL: <https://invisible-island.net/ncurses/ncurses.faq.html>.
- [38] Thomas E. Dickey. *XTerm—Frequently Asked Questions*. URL: <https://invisible-island.net/xterm/xterm.faq.html>.
- [39] “dmesg(1)—print or control the kernel ring buffer.” In: *Linux Programmer’s Manual*.
- [40] “epoll(7)—I/O event notification facility.” In: *Linux Programmer’s Manual*.
- [41] Pavel Fatin. *Typing with pleasure*. Tech. rep. 2015-12. URL: <https://pavelfatin.com/typing-with-pleasure/>.
- [42] Eric Fischer. *The Evolution of Character Codes, 1874–1968*. Tech. rep.
- [43] Unified Extensible Firmware Interface Forum. *UEFI Specification Version 2.8*. Tech. rep. 2019-03.
- [44] Xorg Foundation. “systemd—logind.service, systemd-logind—Login manager.” In: *systemd Manual*.
- [45] N. Freed and J. Postel. *IANA Charset Registration Procedures*. BCP 19. RFC Editor, 2000-10.
- [46] Mike Frysinger. *hterm and Secure Shell: hterm Control Sequences*. 2019. URL: <https://chromium.googlesource.com/apps/libapps/+master/hterm/doc/ControlSequences.md>.
- [47] Nick Gammon. *Character sets, encodings, and Unicode*. Tech. rep. 2017-03. URL: <https://www.gammon.com.au/unicode/>.
- [48] Berny Goodheart. *UNIX Curses Explained*. Prentice Hall, 1991. ISBN: 139319575.
- [49] Manish Goregaokar. *Let’s Stop Ascribing Meaning to Code Points*. Tech. rep. 2017-01. URL: <https://manishearth.github.io/blog/2017/01/14/stop-ascribing-meaning-to-unicode-code-points/>.
- [50] Lexi Summer Hale. *On Terminal Control (everything you ever wanted to know about terminals (but were afraid to ask))*. Tech. rep. URL: <http://xn--rpa.cc/irl/term.html>.
- [51] G. H. Hardy. *A mathematician’s apology*. English. Canto. Cambridge University Press Cambridge, 1992. ISBN: 0521427061. URL: <http://www.loc.gov/catdir/toc/cam029/91036386.html>.
- [52] Michael A. Hiltzik. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*. Harper Business, 2000. ISBN: 0887309895.
- [53] P. Hoffman and F. Yergeau. *UTF-16, an encoding of ISO 10646*. RFC 2781. RFC Editor, 2000-02.
- [54] Mary Ann Horton (as Mark Horton). “The New Curses and Terminfo Package.” In: *USENIX Conference Proceedings* (1982), pp. 79–91.
- [55] IEEE and The Open Group. *The Open Group Base Specifications Issue 7*. Tech. rep. 1003.1-2017. 2018.
- [56] Blue Planet Software Inc. *2009 Tetris® Design Guideline*. Tech. rep. 2009-03.
- [57] Matthew Inman. “Why Nikola Tesla was the greatest geek who ever lived.” In: *The Oatmeal* (2012).
- [58] American National Standards Institute. *Extended Latin Alphabet Coded Character Set for Bibliographic Use (ANSEL)*. Tech. rep. 1993.
- [59] American National Standards Institute. *INCITS 4:1986 [R2002] Information Processing—Coded Character Sets—7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*. Tech. rep. 1986.
- [60] “Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange.” In: TIA-232-F (1997).
- [61] “ioctl_console(2)—ioctls for console terminal and virtual consoles.” In: *Linux Programmer’s Manual*.
- [62] “ioctl_tty(2)—ioctls for terminals and serial lines.” In: *Linux Programmer’s Manual*.
- [63] *Information technology—Procedure for registration of escape sequences and coded character sets*. Standard. Geneva, CH: International Organization for Standardization, 2003-02.

- [64] *Codes for the representation of names of countries and their subdivisions—Part 1: Country codes*. Standard. Geneva, CH: International Organization for Standardization, 2013-11.
- [65] *Information technology—ISO 8-bit code for information interchange—Structure and rules for implementation*. Standard. Geneva, CH: International Organization for Standardization, 1991-12.
- [66] *Information technology—ISO 7-bit coded character set for information interchange*. Standard. Geneva, CH: International Organization for Standardization, 1991-12.
- [67] *Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin alphabet №1*. Standard. Geneva, CH: International Organization for Standardization, 1998-04.
- [68] André Jacques. “ISO Latin-1, norme de codage des caractères européens? Trois caractères français en sont absents!” In: 25 (1996), pp. 65–77. URL: http://cahiers.gutenberg.eu.org/cg-bin/article/CG_1996___25_65_0.pdf.
- [69] Andrew Jenner. *1K colours on CGA: How it’s done*. Tech. rep. 2015-04. URL: <http://www.reenigne.org/blog/1k-colours-on-cga-how-its-done/>.
- [70] Douglas W. Jones. *Punched Cards: A brief illustrated technical history*. URL: <http://homepage.divms.uiowa.edu/~jones/cards/history.html>.
- [71] Ed Jones. *Non 8-bit char support in Clang and LLVM*. Tech. rep. 2017-04. URL: <https://www.embecosm.com/2017/04/18/non-8-bit-char-support-in-clang-and-llvm/>.
- [72] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010. ISBN: 9781593272203.
- [73] Anne van Kesteren. “Encoding Living Standard.” In: (2020). URL: <https://encoding.spec.whatwg.org/>.
- [74] “kqueue(2)—kernel event notification mechanism.” In: *BSD System Calls Manual*. 2018-07.
- [75] Norbert Landsteiner. *Raster CRT Typography (According to DEC)*. 2019-02-20. URL: <https://www.masswerk.at/nougobang/2019/dec-crt-typography>.
- [76] Gilles Leblanc. “What is the exact difference between a ‘terminal’, a ‘shell’, a ‘tty’ and a ‘console’?” In: *Stack Exchange: UNIX & Linux* (2010-11). URL: <https://unix.stackexchange.com/questions/4126/what-is-the-exact-difference-between-a-terminal-a-shell-a-tty-and-a-con/>.
- [77] John Strang Linda Mui Tim O’Reilly. *termcap & terminfo*. O’Reilly Media, 1988-04.
- [78] John MacFarlane. *Pandoc, a universal document converter (Accessed 2020-02-16)*. URL: <https://pandoc.org/MANUAL.html>.
- [79] Charles E. Mackenzie. *Coded Character Sets, History and Development*. Addison-Wesley Publishing Company, 1980. 513 pp.
- [80] Niall Mansfield. *The Joy of X: An Overview of the X Window System*. Addison-Wesley Professional, 1993.
- [81] Nicolas Martin. *On the origins of serial communications and data encoding*. Tech. rep. URL: <http://www.dbase.com/Knowledgebase/dbulletin/bu07sh.htm>.
- [82] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004-07. ISBN: 0201702452.
- [83] Ingo Molnar. “SD still better than CFS for 3d? (was Re: 2.6.23-rc1).” In: *Linux Kernel Mailing List* (2007-07). URL: <https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg190078.html>.
- [84] Randall Munroe. “Up Goer Five.” In: *XKCD* (2012-11). URL: <https://xkcd.com/1133/>.
- [85] Nick Nicholas. *Greek Unicode Issues*. URL: <https://www.opoudjis.net/unicode/unicode.html>.
- [86] Thomas Orozco. *A Deep Dive into the SIGTTIN / SIGTTOU Terminal Access Control Mechanism in Linux*. Tech. rep. 2015-11. URL: <http://curiousthing.org/sigttin-sigtou-deep-dive-linux>.
- [87] Aivosto Oy. “7-bit Character Sets.” In: *Aivosto Articles for developers*. URL: <https://www.aivosto.com/articles/charsets-7bit.html>.
- [88] Jon Peddie. *Famous Graphics Chips: TI TMS34010 and VRAM. The first programmable graphics processor chip*. Tech. rep. URL: <https://www.computer.org/publications/tech-news/chasing-pixels/Famous-Graphics-Chips-IBMs-professional-graphics-the-PGC-and-8514A/Famous-Graphics-Chips-TI-TMS34010-and-VRAM>.
- [89] “posix_openpt(3)—open a pseudoterminal device.” In: *Linux Programmer’s Manual*.
- [90] Ed Post. *Real Programmers Don’t Use PASCAL*. 1992. URL: https://web.mit.edu/humor/Computers/real_programmers.
- [91] Enlightenment Project. *Terminology*. Tech. rep. URL: <https://www.enlightenment.org/about-terminology>.
- [92] FreeVGA Project. “VGA Chipset Reference.” In: (1998-06). URL: <http://www.osdever.net/FreeVGA/home.htm>.

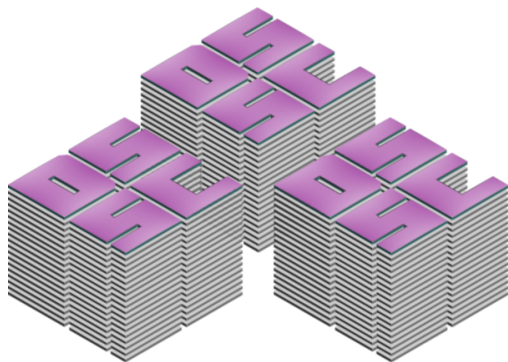
- [124] Z-AXIS. “Boundless Technologies.” In: (2020). URL: <http://www.boundlessterminals.com/>.

Acknowledgments

Hail Eris! All Hail Discordia!

This work would not have been possible without the early customers of Dirty South Supercomputing; thank you Jeff Arnold at ShareCare, Todd Wilson at Vakaros, Charles Brian Quinn at Greenzie, Carl Ledbetter at SimpleRose, Keshav Attrey at PureStorage, Jaron Nix at Pathware, Miguel Turner at Cyxtera, and the folks who can't be mentioned. Keep churning through DGEMMs and blowing shit up, everybody. Yacin Nadji, Scott Hughes, Paige Bailey, Brendan Dolan-Gavitt, Brett W. Thompson, Bill Phillips, Lee Hall, Jon Paprocki, and Joe Lafiosca were particularly supportive of the Notcurses endeavor. Mark Ferrari let me use his mindblowing palette-cycling pixel art, and was awfully sweet to a weird dude emailing him from out of nowhere. GitHub has remained shockingly useful and unoffensive for a Microsoft product. Robert Edmonds helped me get Notcurses into Debian's NEW queue, and coached me in the ways of the DFSG. Marek Habersack wrote the C++ wrapper, and has kept it up to date despite my total lack of communication or warning before lunging in entirely new API directions. Where would I be without jwz? Astrid Bin did the awesome DSSCAW logo¹¹⁰, and I couldn't pay her in the end due to bullshit laws—like, I drunkenly called her up and was like “FUCK DA POLICE i'll drop a brown paper sack of \$10 bills on your porch and steal you a social security number, arrrrrrr” and yet she refused—thanks, Astrid! Mary Ann Horton, the original author of terminfo, was gracious and helpful in her responses. Thomas E. Dickey, author/maintainer of many venerable trees including NCURSES, is a saint and an angel, a UNESCO treasure, and the very model of conservative, thoughtful software stewardship. He graciously answered my mails with lengthy and rigorous information. The free software community owes him a great debt.

Shouts out to Midtown Yuppie Scum (ATL) and Stinkeye of the Tiger (NYC) Trivia Clubs. Shouts out to Paul Johnson and Paul Judge. Shouts out to Outkast, Goodie Mob, and RTJ. Shouts out to every freedom-loving person around the world fighting the unceasing struggle against International Communism. God save the Constitution and God save the Republic.



¹¹⁰Minus the boss purple gradient, which I applied against her passionate wishes. I stand by my call.



About the Author

Aside from adventures in New York and Austin, Nick Black is a lifelong ATLien. He began programming on an ATARI 400 sometime during childhood, and with no one around to tell him better, developed an idiosyncratic, unorthodox style involving more inline assembly and literary allusions than strictly necessary, or perhaps even justifiable. He has since graduated several times from the Georgia Institute of Technology, and dropped out almost as many times. Approaching forty, he still manages to code about ten hours a day, every day, ideally those free of the Daystar's malignant influence. He hopes to one day destroy the sun. This is his first book as an adult. He lives in Midtown Atlanta with wife Emily (both a finer engineer, and just plain finer), along with their poofy penguin and several orca. Nick exclusively uses black IBM Model-M Trackpoint II M-13 keyboards, and will happily buy any that you might have laying around **CLACK CLACK CLACK**.