

Platformatic

View online

Download PDF

Project Destino: Doom In The Terminal with Node.js and OpenTUI

Paolo Insogna

Node.js TSC, Principal Engineer

STFU!



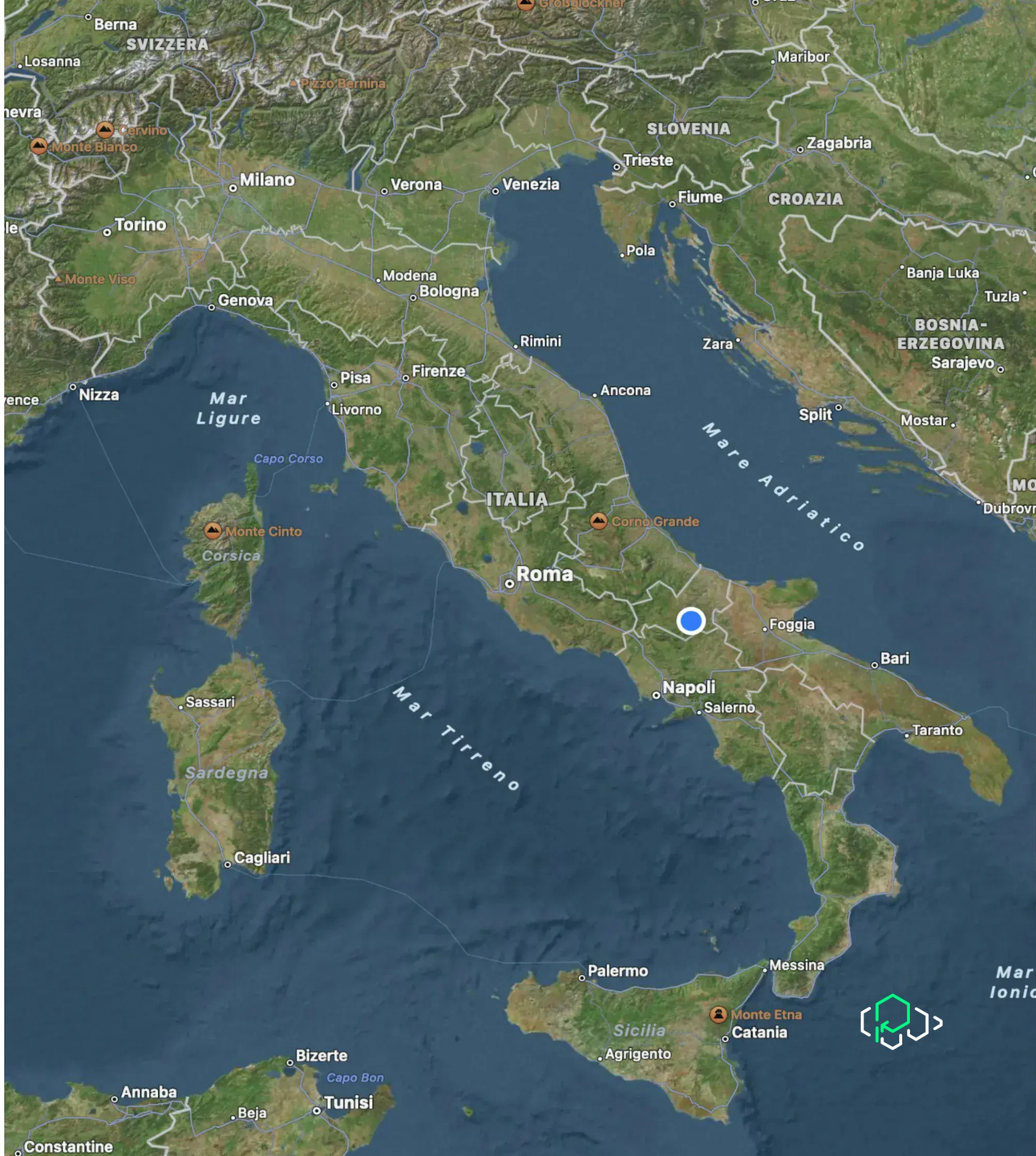
Hello, I'm **Paolo**!



Node.js Technical Steering Committee Member

Platformatic Principal Engineer

`paoloinsogna.dev` `ShogunPanda` `p_insogna` `pinsogna`



Some companies trusting us

Booking.com

Article

supabase

Case Study

SPENDESK

Case Study

This started as a joke...





... and now we have Doom in the terminal!

The joke was simple

1. **Doom can run everywhere.**
   Like a pregnancy test.

2. **Node.js can now run native code.**
   Surprisingly, this was missing.

3. **Why not run Doom in a terminal?**
   I mean, why not?



The terminal is the new UI.

Thanks to AI. 🤷



OpenTUI

A native terminal UI core written in Zig with TypeScript bindings. It powers OpenCode.

https://github.com/anomalyco/opentui

They really push the terminal hard



More than text

Terminal interfaces can combine text and graphics through dedicated protocols.



Refresh rates matter

Dynamic interfaces need redraws that feel immediate.



OpenCode proves the edge

Complex tools can live where terminals usually break.



But this is not only about Doom.



What is FFI?



Foreign Function Interface

A bridge that lets code call functions from another runtime.



C as the common ground

Most native libraries expose symbols using the C calling convention.



Languages can talk

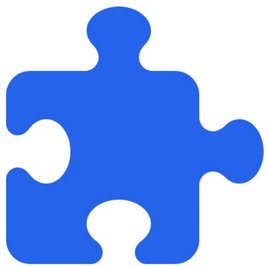
If both sides understand the ABI, JavaScript can call native code.



Before FFI,
we had choices.

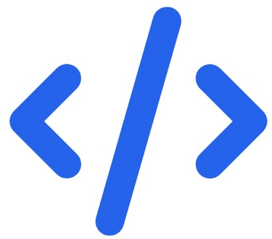


The "choice"



Native addons

Powerful, but abstractions like NaN
used to break with Worker Threads.



Not direct enough

You still had to write a lot
of glue code around the library.



Say hello
to `node:ffi`.



`node:ffi` changes the shape.

Load a shared library, describe the ABI, and call native symbols directly from JavaScript.

```
import { dlopen } from 'node:ffi'

const { functions } = dlopen(libPath, {
  doomgeneric_Tick: { arguments: [], return: 'void' },
  send_key: { arguments: ['uint8', 'int32'], return: 'void' },
  get_framebuffer: { arguments: [], return: 'pointer' }
})
```



The boundary is explicit



Symbols are named

JavaScript points at the native entry points directly.



Types are declared

Arguments and return values become part of the call site.



Risk is visible

The dangerous part is not hidden inside generated glue.

FFI is very powerful.





What can go wrong?

Pretty much everything

And this is because **you** are in full control.



Signature mismatch

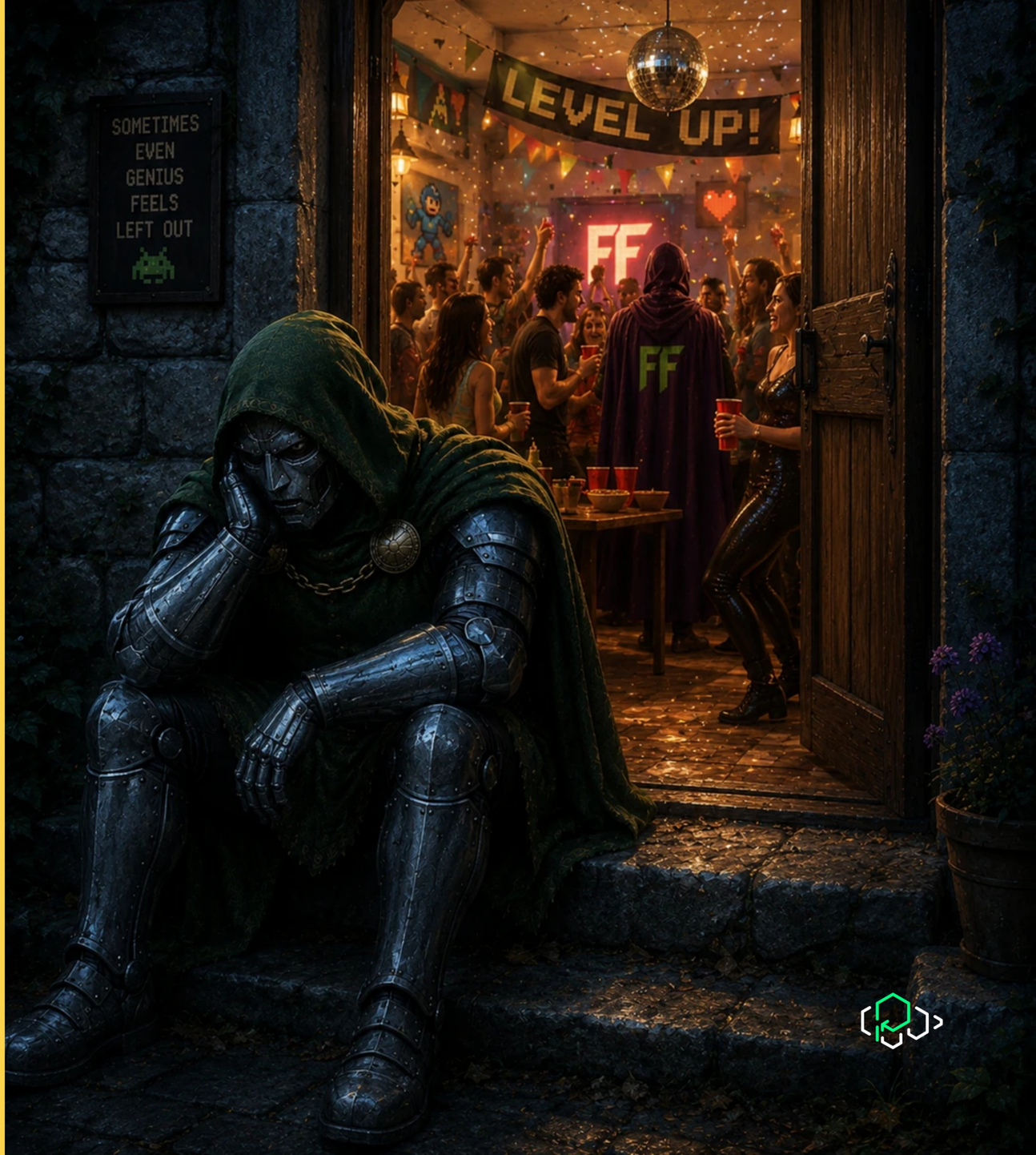
One wrong type can shift registers, stack slots, and return values.



Unsafe memory access

A bad pointer can read freed memory or overwrite valid data.

OK, but what about Doom?



Say hi to Destino.



The Destino stack

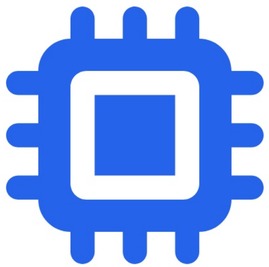
1. **Node.js (https://github.com/nodejs/node)**
   The best JavaScript runtime. **Ever.** 😈 It coordinates native calls through `node:ffi`.

2. **doomgeneric (https://github.com/ozkl/doomgeneric)**
   The game core and native adapters produce frames and audio samples.

3. **OpenTUI + Kitty protocols + SDL3**
   Kitty Graphics presents frames; Kitty keyboard reports key releases; SDL3 plays audio.

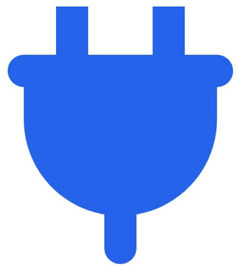


How doomgeneric works



Doom stays portable

The engine calls a tiny platform layer for video, input, timing, and shutdown.



The platform fills the gaps

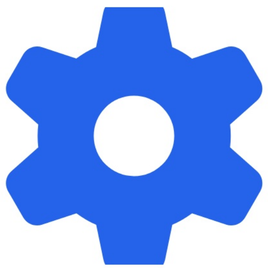
Destino implements that layer as a native library that JavaScript can drive.

Everyone owns a piece



Node.js owns the loop and the loading

It parses input and coordinates ticks, video, audio, and cleanup.



Native code owns the game and audio mixing

The engine and audio backend produce pixels and PCM samples.



OpenTUI and SDL3 handle output

OpenTUI presents frames through Kitty Graphics; SDL3 plays PCM.

Keep the native boundary explicit.



The C platform layer



Initialize and cleanup

Start Doom with assets and release native resources.



Advance ticks and deliver input

Let JavaScript drive the engine and forward key transitions.



Expose pixels and PCM

Give JavaScript a framebuffer and blocks of mixed audio.



The 35 Hz loop

```
// Simplified: error handling and cleanup omitted.
setInterval(() => {
  engine.tick()
  audio.write(engine.renderAudio())
  renderer.render()

  if (engine.quitRequested()) {
    shutdown()
  }
}, 1000 / 35)
```

Now render Doom in a terminal.



Framebuffer to terminal



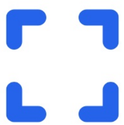
Doom exposes BGRA

Native pixels are exposed through a borrowed Buffer.



JavaScript prepares RGBA

Channels are converted before passing the frame to OpenTUI through FFI.



OpenTUI speaks Kitty Graphics

Native rendering produces the terminal output.



Terminals are weird



Cells are not pixels

Geometry affects aspect ratio and perceived detail.



Layout can change

Resizing and font changes affect how the frame fits.



Negotiate, don't assume

Require Kitty Graphics and query pixel geometry, with no fallback.



Input is the awkward part.



Every game needs input



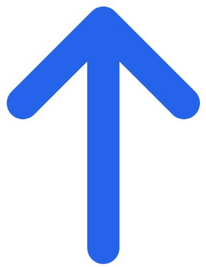
Key press

Movement starts when the key goes down.



Key repeat

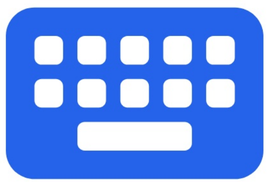
Holding a key must keep the action alive.



Key release

Movement stops when the key comes back up.

Why the Kitty keyboard protocol?



Terminals speak text

Classic input streams are great for characters, not game controls.



Doom needs state

Controls depend on press, repeat, and release events being distinct.



Kitty reports it

Destino verifies press/repeat/release support.

Native mixing. JavaScript orchestration.



Native code produces PCM

TinyMidiLoader and TinySoundFont drive music; the C backend mixes effects.



Samples cross FFI in blocks

Each tick produces 1,260 stereo frames at 44.1 kHz.



Node feeds SDL3

JavaScript queues the PCM and limits backlog without mixing individual samples.

How did we package this monster?



Node.js SEA works, with a catch



SEA provides a virtual filesystem

JavaScript, libraries, and assets ship together with `useVfs`.



Native I/O is a different boundary

C calls to `fopen()` do not automatically use Node's virtual filesystem.

How would you solve that?



Don't forget about K.I.S.S.ing!



Assets in memory. Saves on disk.

1 **Pass assets through FFI**
Node reads WAD and SF2 assets and passes their bytes to native code.

2 **Adapt native file access**
The engine uses memory-backed files in SEA mode.

3 **Persist the saves**
Save files live on disk; other engine writes stay in process memory.



Then comes performance.



Destino is not the hottest case



Doom runs at 35 Hz

The main loop is steady, but not extreme.



Batch work across the boundary

44,100 audio frames per second become 35 PCM blocks, not 44,100 FFI calls.



Real apps can be hotter

Enterprise libraries may sit on critical paths.



(Very) Fast FFI is now in Node.js.

Starting in Node.js 26.4.0

What was added?

Check it out at **https://github.com/nodejs/node/pull/63068**.



V8 Fast API path
Eligible calls avoid the generic callback route.



Native trampolines
Generated code adapts V8 calls to the target ABI.



Safe fallback
Unsupported signatures continue through existing paths.

Why trampolines?



V8 Fast API skips the slow callback

V8 can call a known, typed native function directly.



FFI symbols are discovered at runtime

The target signature is not a handwritten C++ binding.



Trampolines adapt the call

They move arguments from V8's call shape into the target C ABI.



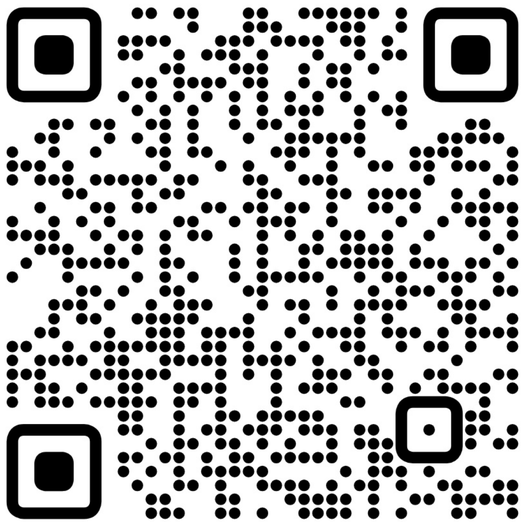
DEMO

TIME



Check it out!

As usual, everything we build at Platformatic is available on GitHub.



https://github.com/platformatic/destino



WHAT'S NEXT?

One last thing™

“The only way of discovering the limits of the possible is to venture a little way past them into the impossible.”

Arthur C. Clarke





Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic