LateralusOS GUI: Framebuffer, Desktop, and Input

A minimal graphical environment for a freestanding 64-bit kernel

Lateralus Language

bad-antics · April 2026 · LateralusOS Architecture Notes

ABSTRACT LateralusOS v0.1 ships with a minimal but complete graphical environment: a double-buffered framebuffer over a linear-mode VESA surface, a compositor with windows and a cursor, and a PS/2 input stack wired through to the application event loop. This note describes each of the three layers as they live in lateralus-os/gui/, covers the boot-time discovery of the framebuffer via the Multiboot2 tag, and shares the debugging notes that brought the system from a blank screen to a renderable desktop in three revisions.

1. Overview

The v0.1 GUI stack has three source files totalling roughly 900 lines of C:

- gui/framebuffer.c — linear-framebuffer primitives (pixel, line, rect, blit, swap).
- gui/desktop.c — compositor with window list, z-ordering, and cursor overlay.
- gui/mouse.c — PS/2 mouse decoder and delta-coalescing event queue.

The kernel is 64-bit freestanding, built with -ffreestanding -nostdlib -mno-red-zone -mgeneral-regs-only and a custom linker script. No standard-library symbols are linked; the framebuffer stack provides its own memset, memcpy, and simple integer-to-string helpers.

2. Framebuffer Discovery

The kernel locates the framebuffer through the Multiboot2 protocol: the bootloader (GRUB2) passes a tag of type FRAMEBUFFER (8) containing physical address, width, height, pitch, and bits-per-pixel. Our boot stub walks the tag list and stores the fields in a global fb_info_t:

```
typedef struct {
    uint64_t addr;      // physical framebuffer address
    uint32_t width;     // pixels
    uint32_t height;    // pixels
    uint32_t pitch;     // bytes per scanline
    uint8_t  bpp;       // bits per pixel (32 in practice)
} fb_info_t;
```

On a typical QEMU -vga std session, the addr lands at 0xFD000000 with a 1024x768x32 mode. Real hardware is wildly variable; we do not assume any specific resolution or address.

3. Page-Table Mapping

The framebuffer physical address is almost never in the identity-mapped low 2 MB that our bootstrap sets up. We extend the page tables in boot_stub.c to cover the full 4 GB physical range with 2 MB pages before enabling long mode:

```
for (uint64_t p = 0; p &lt; 4ULL * 1024 * 1024 * 1024; p += 0x200000) {
    pd[p >> 21] = p | PTE_PRESENT | PTE_WRITE | PTE_PS;
}
```

This is the minimum coverage that reliably includes a VESA framebuffer. Larger systems with framebuffers beyond 4 GB would need dynamic mapping; for the v0.1 target (QEMU and commodity VMs) the 4 GB flat map is correct and cheap.

We spent a debugging cycle chasing a black-screen bug that turned out to be a missing framebuffer mapping: writes to the framebuffer address silently page-faulted, the fault handler was not yet installed,

and the CPU halted with no output. The fix was the loop above; we now install a page-fault handler early enough to catch similar misses in future changes.

4. Double-Buffering

Direct writes to the framebuffer flicker and tear, especially when the compositor is doing more than one layer of work per frame. gui/framebuffer.c allocates a back buffer of equal size in the kernel heap, and all drawing operations target the back buffer. A single fb_swap() call at end-of-frame performs a rep movsq from back to front:

```
void fb_swap(void) {
    size_t qwords = (fb.pitch * fb.height) / 8;
    asm volatile ("rep movsq"
        : "+D"(fb.addr), "+S"(back_buffer), "+c"(qwords)
        : : "memory");
}
```

At 1024x768x32 the copy is 3 MB per frame; at 60 fps that's 180 MB/s sustained, well under the memory bandwidth of anything we run on. We considered write-combining MTRRs for further speedup but found the complexity not justified at the current resolutions.

5. Compositor Model

The desktop maintains a simple window list, each element carrying:

- Position (x, y) and size (w, h) in pixels.
- A content-bitmap pointer (back-buffer region the client drew into).
- A title string (up to 32 bytes, ASCII).
- A z-index; the list is sorted by z on every insertion and raising.

Per-frame rendering: (1) clear the back buffer to the desktop colour, (2) iterate windows in z order blitting each into the back buffer with a 1-pixel border and titlebar, (3) blit the cursor on top at the current mouse position, (4) swap buffers. No dirty-region tracking in v0.1; full redraws at 60 fps are fast enough on the hardware we target.

6. PS/2 Mouse Input

PS/2 is still the lowest-common-denominator pointing device in VMs. Our driver configures the controller for three-byte packets and installs an IRQ 12 handler that decodes packets into (dx, dy, buttons) deltas. The kernel event loop polls the ring buffer and updates the cursor position each frame, clamping to the framebuffer bounds.

```
void mouse_isr(void) {
    mouse_packet[mouse_cycle++] = inb(0x60);
    if (mouse_cycle == 3) {
        mouse_cycle = 0;
        int dx = mouse_packet[1];
        int dy = mouse_packet[2];
        if (mouse_packet[0] &amp;amp; 0x10) dx -= 256;
        if (mouse_packet[0] &amp;amp; 0x20) dy -= 256;
        mouse_event_push(dx, -dy, mouse_packet[0] &amp;amp; 7);
    }
    pic_eoi(12);
}
```

Note the y-inversion: PS/2 reports positive dy for up, we want positive for down.

7. Colour Palette

Even a debug-quality GUI benefits from looking deliberate. LateralusOS v0.1 uses a 4-colour palette evoking the pipeline-first brand identity:

- **Desktop**: #0A0A0F (near-black with a blue tilt).
- **Window body**: #15151C.
- **Window accent**: #FF5CAA (the Lateralus pipeline pink).
- **Cursor**: #F8F8F2 (off-white).

Monospace text on the titlebar uses an embedded 8x16 bitmap font, rendered glyph-at-a-time. The font is baked into the kernel binary as a const uint8_t[].

8. Build and Boot

The complete build is a shell script:

```
./build_and_boot.sh --iso   # builds build/lateralus-os.iso
./build_and_boot.sh --test  # headless QEMU run, serial to build/serial.log
./build_and_boot.sh --gui   # QEMU with SDL/GTK display
```

On any x86_64 host with nasm, gcc, grub-mkrescue, xorriso, and qemu-system-x86_64 installed, the ISO is reproducible from a fresh checkout in roughly 8 seconds.

9. Debugging Story

The GUI stack went through three revisions before rendering stably:

- **Rev 1**: wrote directly to framebuffer; tore on every frame. Lesson: need back buffer.
- **Rev 2**: back buffer in place, but framebuffer address was not mapped; CPU halted silently. Lesson: map the full 4 GB identity range before any MMIO access.
- **Rev 3**: everything mapped, buffer swap working, but cursor leaked trails on mouse movement. Lesson: composite the cursor on top of a full redraw, don't try XOR tricks.

Each revision was a single-digit-hours fix once the symptom was correctly diagnosed; together they spanned two full debug sessions because the symptoms (black screen, halt, cursor trails) were each misleading in a different way.

10. Conclusion

The LateralusOS v0.1 GUI is deliberately minimal: no GPU, no acceleration, no window-server protocol, no compositor-to-client IPC. It exists to prove that a Lateralus-fronted kernel can drive a graphical environment end-to-end in under 1,000 lines of support C, and to give the rest of the OS team a stable surface to build on. Planned v0.2 work: a client-side drawing protocol, input-focus management, and a simple text-input widget so the shell can move out of serial and into a window.

Lateralus is an open-source, zero-dependency programming language. Project home: https://lateralus.dev. Source: github.com/bad-antics/lateralus-lang. Released under CC BY 4.0.