Capítulo 25 — DRM/KMS: Gráficos sin intermediarios

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en file descriptors (`22-file-descriptors`), punteros (`07`), memoria dinámica (`09`) y structs (`08`).*
>
> *Este capítulo te lleva del texto plano al video: escribir píxeles directamente en la pantalla, sin X11, sin Wayland, sin SDL, sin Raylib. Solo Linux + tu GPU + syscalls crudas envueltas por libdrm y libgbm.*

Cómo leer este capítulo

Imaginate que abrís la terminal y escribís `ls` . La salida aparece en una ventana, en una fuente, con colores. Todo eso lo maneja un **servidor de ventanas** (X11 o Wayland). Pero ese servidor no es magia: **alguien** le tiene que decir al kernel "poné esta imagen en la pantalla".

Ese "alguien" es DRM/KMS. Es el subsistema del kernel que controla **directamente** la GPU y los monitores conectados, sin intermediarios. Cuando apagás la laptop y la pantalla se pone negra, es el kernel quitando el scanout (el barrido de píxeles) porque DRM lo dispuso así.

DRM/KMS es **duro**. No hay funciones como `drawCircle()` ni `setResolution()` . Hay ioctls que manejá handles, IDs y punteros crudos a buffers de memoria. Pero una vez que entendés el flujo:

> ***Abrís el dispositivo → enumerateás monitores y modos → creás un buffer → lo mapeás a memoria → dibujás píxeles directamente en RAM → le decís al kernel "mostrá este buffer" → repetís.***

Ese es el ciclo de vida de toda aplicación gráfica en Linux sin windowing system. Todo lo que verán en este capítulo es variaciones sobre ese tema.

Está dividido en **catorce partes**. Después de cada concepto hay una o más prácticas —uno o más ejercicios— para que los uses ya mismo. **No las saltee**: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	¿Qué es DRM/KMS?	El kernel controla tu pantalla; DRM es el camino directo para userspace.
2	Abrir `/dev/dri/cardN`	Cada GPU tiene un fd: abrí el primero disponible y listo.
3	Enumerar recursos	Conectores, CRTC, encoders — todo con `drmModeGetResources`.
4	Conectores y estados	Qué monitor está conectado, tipo de puerto, dimensiones físicas.
5	Modos de pantalla	Resoluciones y refresh rates; elegir uno y aplicarlo a un CRTC.
6	Dumb buffers	Crear buffer del kernel, mapearlo a memoria CPU con `mmap`.
7	Dibujar primitivas en C puro	Bresenham, midpoint, lerp — matemática gráfica sin bibliotecas de renderizado.
8	Framebuffers	Asociar un handle de buffer a un fb_id para que el CRTC lo escanee.
9	Page flip + vblank	Doble buffering: cambiás el buffer visible al próximo vblank, cero tearing.
10	KMS atómico	Configurar CRTCs, modos y planes en UNA sola operación; base del multi-monitor.
11	Eventos DRM y hotplug	`poll()` + handlers para detectar conexión/desconexión de monitores.
12	OpenGL ES + EGL	Contexto gráfico sin windowing system, primitivas con shaders sobre GBM buffers.
13	Vulkan surfaceless	Swapchain surfaceless, pipeline raster, dibujar sin windowing system.
14	Ejercicio jefe: pelota rebotando	N monitores, animación con gradiente arco-by-arco, física de rebote, hotplug.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada ioctl/función, pero* ***los ejercicios no traen la solución****. El enunciado te da el problema; la respuesta la escribís vos.*

PARTE 1 — ¿Qué es DRM/KMS?

DRM = **Direct Rendering Manager**. KMS = **Kernel Mode Setting**. Son dos caras de la misma moneda: el subsistema del kernel que gestiona el hardware gráfico.

1.1 El problema: ¿quién controla la pantalla?

En Windows o macOS, el driver gráfico y el windowing system están estrechamente acoplados. En Linux, todo está separado:

- **El kernel** (DRM) controla el hardware: qué resolución se muestra, qué buffer se escanea, cuándo hacer page flip.
- **El windowing system** (X11/Wayland) gestiona ventanas, input, composición.
- **Las aplicaciones** dibujan en buffers que el compositor (o el kernel directamente) presenta al usuario.

Cuando usás X11 o Wayland, ellos hablan con DRM por vos. Pero si los sacás del medio:

> ***Podés hablarle directo al kernel.*** *Abrís* `/dev/dri/card0` *, enviás ioctls, y el kernel configura tu monitor a la resolución que querés, muestra el buffer que le digas, sin intermediarios.*

Eso es exactamente lo que hacen los display servers (X11/Wayland). Solo que acá lo hacés **vos**.

1.2 Los bloques fundamentales de DRM/KMS

Todo en DRM/KMS se reduce a unos pocos conceptos del kernel:

Concepto	Qué es	Analogía
Connector	El puerto físico (HDMI, DP, VGA)	La toma de corriente
Encoder	Convierte señal digital → señal del conector	El adaptador de corriente
CRTC	Controlador: qué fb y qué modo se escanean	El enchufe en la pared
Mode	Resolución + refresh rate (1920×1080@60)	110V vs 220V
Framebuffer	Buffer de píxeles que el CRTC escanea	La pantalla misma
Dumb buffer	Buffer mapeable por userspace	Los pixels en RAM del CPU

El grafo de relación es:

```
Connector A ──→ Encoder X ──→ [CRTC 0, CRTC 2]
Connector B ──→ Encoder Y ──→ [CRTC 1]
Connector C ──→ Encoder Z ──→ [CRTC 0, CRTC 1, CRTC 2]
```

El flujo mínimo es:

1. Encontrar un connector conectado.
2. Elegir un modo (resolución) del connector.
3. Asociar ese modo a un CRTC compatible.
4. Crear un dumb buffer del tamaño del modo.
5. Crear un framebuffer a partir del dumb buffer.
6. Decirle al CRTC: "mostrá este framebuffer con este modo".
7. Dibujar píxeles en el buffer mapeado → page flip → repetir.

1.3 Las bibliotecas que vamos a usar

Nada de X11, nada de Wayland, nada de SDL. Solo estas:

Biblioteca	Qué hace	Necesaria para
libdrm	Wrappers C de los ioctls de DRM/KMS	Parts 1–9 (todo lo básico)
libgbm	Generic Buffer Manager de Mesa	Part 12 (OpenGL ES)
EGL	Ponte entre OpenGL ES y el sistema de ventanas (o la falta de uno)	Part 12 (OpenGL ES)
Vulkan Loader	Carga los entrypoints de Vulkan en runtime	Part 13 (Vulkan)

> ***libdrm vs ioctls crudos:*** *Podrías hacer* `open("/dev/dri/card0")` *y luego* `ioctl(fd, DRM_IOCTL_..., &args)` *manualmente. Pero libdrm ya hace todo eso con funciones como* `drmModeGetResources()`, `drmModeSetCrtc()`, *etc. Más legible, menos boilerplate, mismo resultado.*

Flags de compilación

```
# Para libdrm
gcc prog.c -ldrm -o prog

# Para OpenGL ES + EGL sobre DRM
gcc prog_egl.c -lGLESv2 -lEGL -ldrm -lgbm -o prog_egl

# Para Vulkan
gcc prog_vulkan.c -lvulkan -ldrm -lgbm -o prog_vulkan
```

> ***Requisito de sistema:*** *Este capítulo solo se puede desarrollar y ejecutar en Linux con un driver DRM (Intel, AMD, Nouveau). En Windows o macOS no existe* `/dev/dri/cardN`*. Si estás en WSL2, la GPU no es accesible directamente desde el subsistema Linux. Necesitás una máquina real, VM con GPU passthrough, o un sistema remoto con KMS habilitado.*

Práctica de la Parte 1

Ejercicio 25.0 — Compilación del proyecto DRM

Conceptos · flags de compilación para libdrm (`-ldrm`), path de dispositivos `/dev/dri/`

Contexto · Antes de escribir código, necesitás que tu entorno tenga las bibliotecas necesarias y sepas dónde está el dispositivo DRM de tu GPU.

Consigna · En tu sistema de desarrollo (Linux con Mesa/intel/nouveau/amdgpu):

1. Verificá qué dispositivos DRM existen: `ls -la /dev/dri/` .
2. Instalá las development packages necesarias (`libdrm-dev` , `libgbm-dev` , `libegl-dev` , `vulkan-loader-dev`).
3. Compilá un programa mínimo con: `gcc prog.c -ldrm` y verificá que no da error de link.

Criterios de aceptación

- ☐ Existe al menos un dispositivo `/dev/dri/card*` .
- ☐ Las development packages están instalados (headers de libdrm).
- ☐ Compila con `gcc prog.c -ldrm` sin errores ni warnings.

PARTE 2 — Abrir `/dev/dri/cardN`

El primer paso es abrir el dispositivo DRM de la GPU que querés usar. Es un file descriptor normal, como abrir cualquier archivo:

```c
#include <fcntl.h>
#include <unistd.h>
#include <drm.h>

int fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);
if (fd < 0) {
    perror("No se pudo abrir /dev/dri/card0");
    return 1;
}

// Ahora fd es tu conexión con la GPU. Todo ioctl va por acá.
```

2.1 ¿Por qué `/dev/dri/card0` ?

Linux expone cada tarjeta gráfica como un **dispositivo de character** en `/dev/dri/` :

Dispositivo	Qué es	Usar para
`/dev/dri/card0`	Primera GPU del sistema	Rendering + display (KMS)
`/dev/dri/card1`	Segunda GPU (si hay)	Lo mismo
`/dev/dri/renderD128`	Node de rendering (no tiene KMS)	Solo Vulkan/OpenGL, no configurar pantalla

Para KMS **necesitás el** `cardN` (master node), que requiere privilegios para modeset. En la práctica:

- Si estás como root o con `CAP_SYS_ADMIN` , abrís directamente `/dev/dri/card0` .
- Si tenés permisos de grupo `video` o si el display server ya te dio acceso, también funciona.
- Para rendering sin KMS (solo OpenGL/Vulkan), usá `renderD128` .

2.2 Verificar capacidades con DRM_GET_CAP

No todos los drivers soportan todo. Podés consultar características del driver con `drmGetCap`:

```c
#include <drm.h>

// ¿Soporta render node?
uint64_t has_renderer = 0;
drmGetCap(fd, DRM_CLIENT_CAP_UNIVERSAL_PLANES, &has_renderer);
printf("Universal planes: %s\n", has_renderer ? "sí" : "no");

// ¿Soporte atómico?
drmGetCap(fd, DRM_CLIENT_CAP_ATOMIC, &has_renderer);
printf("Atomic KMS: %s\n", has_renderer ? "sí" : "no");
```

2.3 Activar el modo atómico

Antes de usar la API atómica (Parte 10), hay que decírselo al kernel con `drmSetClientCap()`:

```c
// Activar modo atómico — hacerlo UNA vez después de abrir el fd
uint64_t atomic = 1;
int ret = drmSetClientCap(fd, DRM_CLIENT_CAP_ATOMIC, atomic);
if (ret) {
    fprintf(stderr, "Atomic no soportado: %s\n", strerror(-ret));
}
```

> ***`DRM_CLIENT_CAP_ATOMIC`:*** *Es una llamada única por fd. Si la hacés ANTES de hacer modeset, el kernel prepara los structs atómicos para todos los ioctls subsecuentes. No tiene efecto si la llamás después de haber hecho un modeset legacy.*

Práctica de la Parte 2

Ejercicio 25.1 — Conectar con la GPU

Conceptos · `open()` en `/dev/dri/card0` , `O_RDWR` , verificación de error, `drmGetCap`

Contexto · Escribí un programa que se conecte a tu GPU y consulte sus capacidades: si soporta universal planes, si soporta modo atómico, y cuántos buffers puede tener el scheduler.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0` con `O_RDWR | O_CLOEXEC` .
2. Si falla, intentar con `/dev/dri/card1` , etc., hasta encontrar uno disponible.
3. Consultar y mostrar: universal planes (`DRM_CLIENT_CAP_UNIVERSAL_PLANES`) y atomic KMS.
4. Activar atomic KMS con `drmSetClientCap()` si aún no está activado.

Entrada · Ninguna (lee los dispositivos disponibles automáticamente).

Salida · Alguna salida similar a:

```
GPU card0 opened successfully
Universal planes: sí
Atomic KMS: activado
```

Identificadores sugeridos `gpuFd` , `cardIndex` , `hasUniversalPlanes` , `hasAtomic`

Criterios de aceptación

- [ ] Abre `/dev/dri/cardN` correctamente (proba con card0, card1, ...).
- [ ] Muestra las capacidades consultadas con `drmGetCap` .
- [ ] Activa atomic KMS con `drmSetClientCap` si no está activado.
- [ ] Maneja el error de apertura (imprime `perror`).
- [ ] Cierra el fd con `close(fd)` al terminar.

PARTE 3 — Enumerar recursos del sistema

Una vez que tenés el fd, necesitás saber qué hay conectado: cuántos monitores, cuántas resoluciones, cuántos controladores de pantalla (CRTC). Todo eso se obtiene con una llamada:

3.1 `drmModeGetResources` — El inventario completo

```c
#include <drm_mode.h>

drmModeRes *resources = drmModeGetResources(fd);
if (!resources) {
    perror("drmModeGetResources falló");
    return 1;
}

// resources->countConnectors       → cuántos conectores hay
// resources->connector_id_ptr      → array de IDs de conector (o NULL)
// resources->countCrtcs            → cuántos CRTC hay
// resources->crtc_id_ptr           → array de IDs de CRTC (o NULL)
// resources->countEncoders         → cuántos encoders hay
// resources->encoder_id_ptr        → array de IDs de encoder (or NULL)

printf("Connectors: %d\n", resources->countConnectors);
printf("CRTCs:      %d\n", resources->countCrtcs);
printf("Encoders:   %d\n", resources->countEncoders);

// Liberar cuando terminás
drmModeFreeResources(resources);
```

> `drmModeGetResources` ***devuelve un bloque de memoria con arrays internos.*** *No son punteros sueltos: el kernel llena una estructura y todos los sub-arrays (connectors, crtcs, encoders) apuntan a memoria dentro del mismo bloque. Por eso* `drmModeFree(resources)` *los libera TODOS juntos.*

3.2 El grafo de conexión: connector → encoder → crtc

No todos los conectores se pueden conectar a todos los CRTCs. Cada conector tiene un **encoder** asociado, y cada encoder puede conectarse a un subconjunto de CRTCs. La relación es:

```
Connector A → Encoder X → [CRTC 0, CRTC 2]
Connector B → Encoder Y → [CRTC 1]
Connector C → Encoder Z → [CRTC 0, CRTC 1, CRTC 2]
```

Esto significa: no podés arbitrariamente asignar cualquier monitor a cualquier CRTC. Necesitás respetar las compatibilidades del hardware. Podés consultar qué CRTCs puede usar un encoder así:

```c
// Obtener info de un encoder específico
drmModeEncoder *enc = drmModeGetEncoder(fd, connector->encoder_id);
printf("CRTC posible: %x\n", enc->possible_crtcs);  // bitmask

// Verificar si el CRTC N es compatible:
if (enc->possible_crtcs & (1 << crtcIndex)) {
    printf("CRTC %d es compatible\n", crtcIndex);
}

drmModeFreeEncoder(enc);
```

3.3 Mapear conectores a CRTCs compatibles

Para cada connector conectado, necesitás encontrar un CRTC libre y compatible:

```
// Para cada conector...
for (int i = 0; i < resources->countConnectors; i++) {
    uint32_t conn_id = resources->connector_id_ptr[i];

    // Obtener info completa del connector (incluye sus encoders)
    drmModeConnector *conn = drmModeGetConnector(fd, conn_id);

    if (conn->connection == DRM_MODE_CONNECTED) {
        printf("Conector %d conectado\n", conn_id);

        // Para cada encoder asociado...
        for (int e = 0; e < conn->count_encoders; e++) {
            drmModeEncoder *enc = drmModeGetEncoder(fd, conn->encoders[e]);

            // Para cada CRTC posible en este encoder...
            for (int c = 0; c < resources->countCrtcs; c++) {
                if (enc->possible_crtcs & (1 << c)) {
                    printf("  → CRTC %d compatible\n", resources->crtc_id_ptr[c]);
                }
            }

            drmModeFreeEncoder(enc);
        }
    }

    drmModeFreeConnector(conn);
}
```

***`DRM_MODE_CONNECTED`** : Es el estado del connector. Puede ser:*

- *`DRM_MODE_CONNECTED` — hay algo enchufado.*
- *`DRM_MODE_DISCONNECTED` — nada conectado.*
- *`DRM_MODE_UNKNOWNCONNECTION` — el kernel no puede determinar.*

Práctica de la Parte 3

Ejercicio 25.2 — Inventario del sistema gráfico

Conceptos · `drmModeGetResources`, arrays de IDs, iteración sobre conectores/CRTC/encoders

Contexto · Escribí un programa que haga un inventario completo del subsistema DRM/KMS: qué monitores están conectados, qué CRTCs hay disponibles y qué compatibilidades existen entre ellos.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0` y obtener recursos con `drmModeGetResources`.
2. Iterar todos los conectores e imprimir su tipo (HDMI, DP, VGA...) y estado (conectado/desconectado).
3. Para cada connector conectado, mostrar sus encoders asociados y qué CRTCs son compatibles.
4. Imprimir un resumen como:

```
HDMI-A-1  → conectado → Encoder 12 → CRTC [0, 2]
DP-1      → desconectado
```

Entrada · Ninguna (detecta automáticamente).

Salida · Listado de conectores, estados y compatibilidades de CRTC.

Identificadores sugeridos `resources`, `connectors`, `crtcIds`, `encoderIds`, `compatibleCrtcs`

Criterios de aceptación

- ☐ Usa `drmModeGetResources` correctamente (con `drmModeFreeResources`).
- ☐ Itera todos los conectores e imprime tipo + estado.
- ☐ Para cada conector conectado, muestra encoders y CRTCs compatibles.
- ☐ Limpia toda la memoria liberada con `drmModeFreeConnector` / `drmModeFreeEncoder`.

PARTE 4 — Conectores y estados

Cada connector representa un puerto físico en tu GPU: HDMI, DisplayPort, VGA, eDP (laptop). DRM/KMS te da información detallada sobre cada uno.

4.1 Consultar un connector

```
drmModeConnector *conn = drmModeGetConnector(fd, connector_id);
if (!conn) {
    perror("drmModeGetConnector falló");
    return 1;
}

printf("Tipo:          %d\n", conn->connector_type);
printf("Subtipo:       %d\n", conn->connector_type_id);
printf("Estado:        %d\n", conn->connection);     // DRM_MODE_CONNECTED o
DRMMODE_DISCONNECTED
printf(" mmWidth:      %d\n", conn->mm_width);         // ancho en mm (para DPI)
printf(" mmHeight:     %d\n", conn->mm_height);        // alto en mm

// Tipo de conector (DRM_MODE_CONNECTOR_*):
switch (conn->connector_type) {
    case DRM_MODE_CONNECTOR_VGA:       printf("VGA\n"); break;
    case DRM_MODE_CONNECTOR_DVII:      printf("DVI-I\n"); break;
    case DRM_MODE_CONNECTOR_DVID:      printf("DVI-D\n"); break;
    case DRM_MODE_CONNECTOR_DisplayPort: printf("DisplayPort\n"); break;
    case DRM_MODE_CONNECTOR_HDMIA:     printf("HDMI-A\n"); break;
    case DRM_MODE_CONNECTOR_eDP:       printf("eDP (laptop)\n"); break;
    case DRM_MODE_CONNECTOR_LVDS:      printf("LVDS (laptop LCD)\n"); break;
    // ... más tipos
}

drmModeFreeConnector(conn);
```

4.2 Tipos de conector más comunes

Constante	Valor	Descripción
`DRM_MODE_CONNECTOR_Unknown`	0	No se pudo identificar
`DRM_MODE_CONNECTOR_VGA`	1	VGA analógico
`DRM_MODE_CONNECTOR_DVID`	3	DVI-D digital
`DRM_MODE_CONNECTOR_HDMIA`	11	HDMI tipo A (estándar)
`DRM_MODE_CONNECTOR_HDMIB`	12	HDMI tipo B (ancha)
`DRM_MODE_CONNECTOR_DisplayPort`	10	DisplayPort
`DRM_MODE_CONNECTOR_eDP`	14	Embedded DP (pantalla de laptop)

4.3 El estado de conexión

El campo `connection` puede tener estos valores:

Estado	Significado
`DRM_MODE_CONNECTED`	Hay algo enchufado y activo
`DRM_MODE_DISCONNECTED`	Nada enchufado
`DRM_MODE_UNKNOWNCONNECTION`	El kernel no sabe (hardware defectuoso o driver limitado)

> ***Hotplug:*** *Cuando conectás o desconectás un monitor, el kernel envía un evento. Tu programa debe estar escuchando eventos de DRM (`poll()` / `epoll()` en el fd) para reaccionar a cambios en la configuración de monitores. Lo veremos más adelante (Parte 11).*

Práctica de la Parte 4

Ejercicio 25.3 — Detector de monitores conectados

Conceptos · `drmModeGetConnector`, tipos de conector, estado de conexión

Contexto · Escribí un programa que detecte qué monitores están físicamente conectados a tu GPU y muestre información detallada de cada uno: tipo de puerto, resolución preferida (si la hay), dimensiones en milímetros.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0` y obtener recursos con `drmModeGetResources`.
2. Iterar todos los conectores.
3. Para cada conector, imprimir: ID, tipo (texto legible), subtipo, estado de conexión y dimensiones físicas.
4. Solo mostrar conectores con estado `DRM_MODE_CONNECTED`.

Entrada · Ninguna.

Salida · Ejemplo:

```
[Conector] ID=32  Tipo=HDMI-A  Subtipo=0  Estado=conectado  598x336 mm
```

Identificadores sugeridos `resources`, `connId`, `connectorType`, `connectedMonitors`

Criterios de aceptación

- [ ] Detecta automáticamente qué conectores están conectados.
- [ ] Muestra tipo legible (HDMI-A, DisplayPort, etc.).
- [ ] Incluye dimensiones físicas en mm del conector.
- [ ] Solo imprime conectores `DRM_MODE_CONNECTED`.
- [ ] Limpia memoria correctamente (`drmModeFreeConnector`, `drmModeFreeResources`).

PARTE 5 — Modos de pantalla

Cada conector conectado tiene una lista de **modos** disponibles: combinaciones de resolución, refresh rate y sincronización. Por ejemplo: 1920×1080@60Hz, 1280×720@60Hz, etc.

5.1 Los modos de un connector

Cuando consultás un connector con `drmModeGetConnector`, el kernel devuelve todos los modos que ese monitor soporta (incluidos los EDID):

```
drmModeConnector *conn = drmModeGetConnector(fd, connector_id);
if (conn->connection != DRM_MODE_CONNECTED) {
    // Nada enchufado — no hay modo válido que aplicar.
    goto cleanup;
}

printf("Modos disponibles (%d):\n", conn->count_modes);
for (int i = 0; i < conn->count_modes; i++) {
    drmModeModeInfo *mode = &conn->modes[i];

    printf("  [%s] %dx%d  @ %.2f Hz  ",
           mode->type & DRM_MODE_TYPE_PREFERRED ? "*" : " ",
           mode->hdisplay, mode->vdisplay,
           mode->clock / 1000.0);

    // Flags de sincronización
    if (mode->flags & DRM_MODE_FLAG_PHSYNC) printf("+H ");
    if (mode->flags & DRM_MODE_FLAG_NHSYNC) printf("-H ");
    if (mode->flags & DRM_MODE_FLAG_PVSYNC) printf("+V ");
    if (mode->flags & DRM_MODE_FLAG_NVSYNC) printf("-V ");

    printf("\n");
}

// El modo preferido es el que el monitor recomienda:
drmModeModeInfo *preferred = NULL;
for (int i = 0; i < conn->count_modes; i++) {
    if (conn->modes[i].type & DRM_MODE_TYPE_PREFERRED) {
        preferred = &conn->modes[i];
        break;
    }
}

if (preferred) {
    printf("Modo preferido: %s (%dx%d)\n",
           preferred->name, preferred->hdisplay, preferred->vdisplay);
}

drmModeFreeConnector(conn);
```

> ***`DRM_MODE_TYPE_PREFERRED`*** *: Es el modo que el monitor reporta como óptimo (vía EDID). Es la resolución nativa del panel. Siempre preferí este a menos que haya una razón específica para elegir otro.*

5.2 El campo `hdisplay` × `vdisplay`

Cada modo tiene estas dimensiones:

Campo	Descripción	Ejemplo (1920×1080@60)
`hdisplay`	Píxeles horizontales visibles	1920
`vdisplay`	Píxeles verticales visibles	1080
`hsync_start`	Inicio de sync horizontal	2008
`hsync_end`	Fin de sync horizontal	2052
`htotal`	Total horizontal (incluyendo blanking)	2200
`clock`	Pixel clock en kHz	148500 (= 148.5 MHz)
`name`	Nombre descriptivo (ej: "1920x1080")	`"1920x1080"`

5.3 Aplicar un modo a un CRTC (modo legacy)

Una vez que elegiste un conector conectado y un modo, necesitás aplicarlo. El **modo legacy** es la forma más simple:

```
// Paso 1: Crear un framebuffer temporal (necesario incluso para inicializar el display).
drmModeFB fb;
if (drmModeAddFB(fd, mode->hdisplay, mode->vdisplay, 24, 32,
                 mode->hdisplay * 4, handle, &fb.id)) {
    perror("drmModeAddFB falló");
}

// Paso 2: Establecer el modo en un CRTC compatible.
uint32_t crtc_id = <elegir_crtc_compatible(conn, resources)>;

if (drmModeSetCrtc(fd, crtc_id, fb.id, 0, 0,
                   &connector_id, 1, mode)) {
    perror("drmModeSetCrtc falló");
}

// Paso 3: Liberar el framebuffer temporal.
drmModeRmFB(fd, fb.id);
```

`drmModeSetCrtc`: *Es la función "haz todo" del modo legacy:*

- *Asigna un CRTC a un connector.*
- *Establece resolución (modo).*
- *Establece el framebuffer visible.*
- *Posiciona el fb en el CRTC (x=0, y=0 para pantalla completa).*

Elegir CRTC compatible

Para elegir el CRTC correcto, usamos la técnica de la Parte 3:

```
uint32_t find_compatible_crtc(int fd, uint32_t connector_id,
                              drmModeRes *resources) {
    drmModeConnector *conn = drmModeGetConnector(fd, connector_id);
    if (!conn || conn->encoder_id == 0) {
        goto err;
    }

    drmModeEncoder *enc = drmModeGetEncoder(fd, conn->encoder_id);
    if (!enc) {
        goto err;
    }

    // Buscar el primer CRTC compatible.
    for (int i = 0; i < resources->countCrtcs; i++) {
        uint32_t crtc_id = resources->crtc_id_ptr[i];

        // ¿Este CRTC es compatible con el encoder?
        if (!(enc->possible_crtcs & (1 << i))) continue;

        // Está disponible — lo usamos.
        return crtc_id;
    }

err:
    if (conn) drmModeFreeConnector(conn);
    if (enc)  drmModeFreeEncoder(enc);
    return resources->crtc_id_ptr[0];  // fallback
}
```

Práctica de la Parte 5

Ejercicio 25.4 — Listar resoluciones por monitor

Conceptos · `drmModeGetConnector`, modos (`drmModeModeInfo`), `DRM_MODE_TYPE_PREFERRED`

Contexto · Escribí un programa que liste todas las resoluciones disponibles para cada monitor conectado, marcando cuál es la resolución preferida (nativa) del display.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0` y obtener recursos.
2. Para cada conector conectado, listar todos sus modos (resoluciones).
3. Marcar con `*` el modo preferido (`DRM_MODE_TYPE_PREFERRED`).
4. Mostrar para cada modo: nombre, resolución, refresh rate estimado, flags de sync.

Entrada · Ninguna.

Salida · Ejemplo:

```
=== Conector HDMI-A-1 (ID=32) — conectado ===
* 1920x1080    @ 60.0 Hz   +H +V
   1280x720     @ 60.0 Hz   +H +V
   1024x768     @ 60.0 Hz   +H +V
   800x600      @ 60.3 Hz   +H +V
```

Identificadores sugeridos `resources` , `connId` , `modeCount` , `preferredMode`

Criterios de aceptación

- ☐ Solo procesa conectores `DRM_MODE_CONNECTED` .
- ☐ Lista todos los modos con nombre, resolución y refresh rate.
- ☐ Marca el modo preferido con `*` .
- ☐ Muestra flags de sincronización (+H/+V o -H/-V).

PARTE 6 — Dumb buffers

Una vez que tenés un monitor configurado (modo aplicado a un CRTC), necesitás mostrar algo en él. El primer paso: pedirle al kernel un buffer de video. En DRM/KMS, esto se llama **dumb buffer**.

6.1 ¿Qué es un dumb buffer?

Un dumb buffer es un bloque de memoria del kernel que representa píxeles en VRAM (o gart). Se llama "dumb" (tonto) porque el kernel lo administra todo: asignación, deallocation, y mapeo. Vos solo Lees/Escribes los bytes cuando está mapeado.

El flujo es:

1. Pedir al kernel un buffer de tamaño w×h×bpp con `DRM_IOCTL_MODE_CREATE_DUMB`.
2. Obtener un **handle** (uint32_t) que identifica el buffer.
3. Mapearlo a memoria CPU con `DRM_IOCTL_MODE_MAP_DUMB` + `mmap()`.
4. Escribir píxeles directamente en el puntero resultante.

```c
#include <drm.h>
#include <drm_mode.h>
#include <sys/mman.h>

// Estructura para crear un dumb buffer
struct dumb_buffer {
    uint32_t handle;
    uint32_t pitch;       // bytes por fila (puede ser > width*bytes_per_pixel)
    uint32_t size;        // tamaño total en bytes
    void     *addr;       // puntero mapeado a memoria CPU
};

// Paso 1: Crear el dumb buffer
struct drm_mode_create_dumb create = {0};
create.width  = mode->hdisplay;
create.height = mode->vdisplay;
create.bpp    = 32;      // 32 bits por píxel (RGBA)

int ret = ioctl(fd, DRM_IOCTL_MODE_CREATE_DUMB, &create);
if (ret < 0) {
    perror("DRM_IOCTL_MODE_CREATE_DUMB falló");
}

struct dumb_buffer dbuf;
dbuf.handle = create.handle;
dbuf.pitch  = create.pitch;
dbuf.size   = create.size;

// Paso 2: Mapear el buffer a memoria CPU
struct drm_mode_map_dumb map = {0};
map.handle = dbuf.handle;

ret = ioctl(fd, DRM_IOCTL_MODE_MAP_DUMB, &map);
if (ret < 0) {
    perror("DRM_IOCTL_MODE_MAP_DUMB falló");
}

// Paso 3: mmap() — esto es un offset físico del driver, NO un offset de archivo
dbuf.addr = mmap(NULL, dbuf.size, PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, map.offset);
if (dbuf.addr == MAP_FAILED) {
```

```
    perror("mmap falló");
}

// ¡Ahora podés escribir píxeles!
printf("Dumb buffer: handle=%u pitch=%u size=%u\n",
       dbuf.handle, dbuf.pitch, dbuf.size);
```

> ***`pitch` vs `width * bytes_per_pixel`:*** *El pitch es la cantidad de bytes por fila del buffer. Puede ser mayor que el ancho real por alineación (por ejemplo, cada fila debe empezar en un boundary de 64 bytes). Siempre usá* `pitch`*, nunca calcules* `y * width * 4` *manualmente.*

6.2 Escribir píxeles RGB888 en el buffer

Para escribir un pixel en la posición `(x, y)` con profundidad 32bpp (RGBA):

```
// Puntero al primer byte del píxel en (x, y)
uint32_t *row = (uint32_t *)((char *)dbuf.addr + y * dbuf.pitch);
uint32_t *pixel = &row[x];

// Escribir color ARGB (little-endian: ABGR en memoria)
*pixel = 0xFF0000FF;  // Rojo puro (en formato que tu driver espere)

// Para 24bpp (RGB, sin alpha):
uint8_t *byte_row = (uint8_t *)dbuf.addr + y * dbuf.pitch;
uint8_t *pixel_byte = &byte_row[x * 3];
pixel_byte[0] = 0xFF;  // R
pixel_byte[1] = 0x00;  // G
pixel_byte[2] = 0x00;  // B
```

> ***Formato de píxel:*** *El formato exacto depende del driver. Los más comunes son:*
>
> - *XRGB8888 (32bpp, alpha ignorado) — el más universal en Linux con Mesa.*
> - *XBGR8888 (32bpp, canal B y R invertidos).*
> - *RGB565 (16bpp, 2 bytes por píxel). Para saber cuál soporta tu driver, consultá* `drmModeGetFB2()` *o usá GBM buffers (Parte 12).*

6.3 Limpieza

Al terminar de usar el dumb buffer, necesitás liberarlo en orden:

```
// 1. Desmapear la memoria
munmap(dbuf.addr, dbuf.size);

// 2. Destruir el buffer del kernel
struct drm_mode_destroy_dumb destroy = {0};
destroy.handle = dbuf.handle;
ioctl(fd, DRM_IOCTL_MODE_DESTROY_DUMB, &destroy);
```

> ***Orden importa:*** *primero* `munmap()` *, después* `destroy` *. Si destruyés antes de desmapear, el kernel libera la VRAM y cualquier acceso futuro es un use-after-free.*

Práctica de la Parte 6

Ejercicio 25.5 — Crear y mapear un dumb buffer

Conceptos · `DRM_IOCTL_MODE_CREATE_DUMB` , `DRM_IOCTL_MODE_MAP_DUMB` , `mmap()` , `munmap()`

Contexto · Escribí un programa que cree un dumb buffer de 640×480 a 32bpp, lo mapee a memoria CPU e imprima sus propiedades: handle, pitch, size y dirección de memoria mapeada.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0` y activar atomic KMS con `drmSetClientCap` .
2. Crear un dumb buffer de 640×480 a 32bpp con `DRM_IOCTL_MODE_CREATE_DUMB` .
3. Mapearlo con `DRM_IOCTL_MODE_MAP_DUMB` + `mmap()` .
4. Imprimir las propiedades del buffer.
5. Liberar en orden: `munmap()` → `DRM_IOCTL_MODE_DESTROY_DUMB` .

Criterios de aceptación

- [ ] Crea un dumb buffer de 640×480 a 32bpp sin errores.
- [ ] Mapea el buffer con mmap() correctamente (no devuelve MAP_FAILED).
- [ ] Imprime handle, pitch, size y dirección base del mapeo.
- [ ] Limpia en orden: munmap → destroy.
- [ ] Maneja errores en cada ioctl (imprime `perror`).

PARTE 7 — Dibujar primitivas en C puro

Aquí empieza la parte divertida: **dibujar formas geométricas directamente en el buffer**. Sin bibliotecas de gráficos, sin OpenGL. Solo matemáticas y punteros crudos a memoria.

7.1 El canvas: un array de bytes

Imagina que el dumb buffer es una cuadrícula infinita de píxeles. Cada píxel tiene 4 bytes (RGBA). Para escribir en la posición `(x, y)`:

```
// Puntero base al buffer mapeado
uint8_t *buf = (uint8_t *)dbuf.addr;

// Calcular offset del píxel en (x, y):
int offset = y * dbuf.pitch + x * 4;

// Escribir un color ARGB (little-endian)
buf[offset + 0] = 255;  // A (alpha, 255 = opaco)
buf[offset + 1] = 0;    // R (rojo)
buf[offset + 2] = 0;    // G (verde)
buf[offset + 3] = 0;    // B (azul)

// En little-endian, el uint32 que escribís es ABGR en memoria:
uint32_t *pixels = (uint32_t *)dbuf.addr;
pixels[y * (dbuf.pitch / 4) + x] = 0xFF0000FF;  // ARGB: rojo en XRGB8888
```

> ***IMPORTANTE:*** *`pitch` está en bytes, no en píxeles. Si usás un array de `uint32_t`, necesitás dividir `pitch` por 4 para obtener el stride en píxeles.*

7.2 Algoritmo de Bresenham: trazado de líneas

Bresenham es el algoritmo clásico para trazar una línea recta entre dos puntos usando solo **sumas, restas y multiplicaciones enteras**. Sin `float`, sin divisiones, sin `sqrt()`.

```
// Dibuja una línea de (x0,y0) a (x1,y1) en un dumb buffer 32bpp ARGB
void draw_line(int fd, uint8_t *buf, uint32_t pitch,
               int x0, int y0, int x1, int y1, uint32_t color) {
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);

    // Dirección en cada eje (+1 o -1)
    int sx = (x0 < x1) ? 1 : -1;
    int sy = (y0 < y1) ? 1 : -1;

    int err = dx - dy;  // Error acumulado

    while (1) {
        // Escribir pixel en (x0, y0)
        uint32_t *pixel = (uint32_t *)(buf + y0 * pitch + x0 * 4);
        *pixel = color;

        // ¿Llegamos al final?
        if (x0 == x1 && y0 == y1) break;

        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 <  dx) { err += dx; y0 += sy; }
    }
}
```

Cómo funciona Bresenham: *El algoritmo mantiene un "error acumulado" (`err`). En cada paso, decide si avanza en X, en Y, o en ambos según si el error supera cierto umbral. El resultado es una línea con píxeles lo más cerca posible de la recta ideal — **sin punto flotante**.*

7.3 Midpoint circle: círculos perfectos

Para dibujar círculos, usamos el algoritmo midpoint (también llamado "Bresenham para círculos"):

```
void draw_circle(int fd, uint8_t *buf, uint32_t pitch,
                 int cx, int cy, int radius, uint32_t color) {
    int x = radius;
    int y = 0;
    int err = 1 - x;  // Error inicial

    while (x >= y) {
        // Dibujar los 8 píxeles de simetría
        draw_pixel_8way(buf, pitch, cx, cy, x, y, color);

        y++;
        if (err < 0) {
            err += 2 * y + 1;
        } else {
            x--;
            err += 2 * (y - x) + 1;
        }
    }
}

// Dibuja un píxel en las 8 posiciones simétricas del círculo
void draw_pixel_8way(uint8_t *buf, uint32_t pitch,
                     int cx, int cy, int x, int y, uint32_t color) {
    // Los 8 octantes
    draw_pixel(buf, pitch, cx + x, cy + y, color);
    draw_pixel(buf, pitch, cx - x, cy + y, color);
    draw_pixel(buf, pitch, cx + x, cy - y, color);
    draw_pixel(buf, pitch, cx - x, cy - y, color);
    draw_pixel(buf, pitch, cx + y, cy + x, color);
    draw_pixel(buf, pitch, cx - y, cy + x, color);
    draw_pixel(buf, pitch, cx + y, cy - x, color);
    draw_pixel(buf, pitch, cx - y, cy - x, color);
}

// Dibuja un solo píxel con protección de boundaries
void draw_pixel(uint8_t *buf, uint32_t pitch, int x, int y, uint32_t color) {
    // Protección simple: no dibujar fuera del buffer
    if (x < 0 || x >= 640 || y < 0 || y >= 480) return;

    uint32_t *pixel = (uint32_t *)(buf + y * pitch + x * 4);
    *pixel = color;
}
```

> ***Simetría de 8 vías:*** *Un círculo es simétrico en los 8 octantes. Cuando calculamos un punto* `(x, y)` *desde el centro* `(cx, cy)`*, automáticamente tenemos 7 puntos más:* `(-x, y)`, `(x, -y)`, `(-x, -y)`, `(y, x)`, `(-y, x)`, `(y, -x)`, `(-y, -x)`.

7.4 Midpoint ellipse: elipses

Para elipses, el algoritmo midpoint se divide en **dos regiones** porque la pendiente cambia:

```c
void draw_ellipse(int fd, uint8_t *buf, uint32_t pitch,
                  int cx, int cy, int rx, int ry, uint32_t color) {
    // Región 1: donde |dy/dx| < 1 (más horizontal que vertical)
    long long x = rx * rx;
    long long y = 0;
    long long err1 = x - (long long)rx * ry + (y * y) / 4;

    while (x > (long long)rx * rx * y / (2 * ry)) {  // Región 1
        draw_pixel_8way(buf, pitch, cx, cy,
                        lround(x / rx), lround(y / ry), color);
        y++;
        err1 += (2 * y - 1) * x;
        if (err1 >= 0) {
            x--;
            err1 -= (long long)ry * ry * 2;
        }
    }

    // Región 2: donde |dy/dx| >= 1 (más vertical que horizontal)
    x = 0;
    y = ry * ry;
    long long err2 = y - (long long)rx * ry + (x * x) / 4;

    while (y > (long long)ry * ry * x / (2 * rx)) {  // Región 2
        draw_pixel_4way(buf, pitch, cx, cy, lround(x / rx), lround(y / ry), color);
        x++;
        err2 += (2 * x - 1) * y;
        if (err2 >= 0) {
            y--;
            err2 -= (long long)rx * rx * 2;
        }
    }
}

void draw_pixel_4way(uint8_t *buf, uint32_t pitch,
                     int cx, int cy, int x, int y, uint32_t color) {
    draw_pixel(buf, pitch, cx + x, cy + y, color);
    draw_pixel(buf, pitch, cx - x, cy + y, color);
    draw_pixel(buf, pitch, cx + x, cy - y, color);
    draw_pixel(buf, pitch, cx - x, cy - y, color);
}
```

> ***Dos regiones:*** *La mitad superior delipse tiene pendiente < 1 (cambia más en X que en Y), la inferior tiene pendiente >= 1. Por eso el algoritmo se divide: cambia la variable de decisión según qué eje avanza más rápido.*

7.5 Interpolación lineal (lerp) y gradientes

Para colorear formas con gradientes, necesitás **interpolación lineal** entre dos colores:

```
// Lerp entre dos valores (o componentes de color)
int lerp(int a, int b, float t) {
    return a + (int)((b - a) * t);
}

// Interpolar entre dos colores RGB
uint32_t lerp_color(uint8_t r1, uint8_t g1, uint8_t b1,
                    uint8_t r2, uint8_t g2, uint8_t b2, float t) {
    uint8_t r = lerp(r1, r2, t);
    uint8_t g = lerp(g1, g2, t);
    uint8_t b = lerp(b1, b2, t);

    // ARGB (little-endian): ABGR en memoria para XRGB8888
    return ((uint32_t)0xFF << 24) |       // Alpha = 255 (opaco)
           ((uint32_t)b << 16) |          // Blue
           ((uint32_t)g << 8) |           // Green
           ((uint32_t)r);                 // Red
}

// Gradiente lineal de un color a otro a lo largo del ancho de pantalla
void draw_gradient_h(uint8_t *buf, uint32_t pitch, int width, int height,
                     uint8_t r1, uint8_t g1, uint8_t b1,
                     uint8_t r2, uint8_t g2, uint8_t b2) {
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float t = (float)x / (width - 1);
            uint32_t color = lerp_color(r1, g1, b1, r2, g2, b2, t);

            uint32_t *pixel = (uint32_t *)(buf + y * pitch + x * 4);
            *pixel = color;
        }
    }
}
```

> `lerp` ***es la función más importante de gráficos:*** *Interpola entre cualquier par de valores. Se usa para: colores, coordenadas UV, transparencias, animaciones, texturas... casi todo lo que se mueve suavemente en una pantalla pasa por un* `lerp` *.*

Práctica de la Parte 7

Ejercicio 25.6 — Lienzo con primitivas geométricas

Conceptos · Bresenham line, midpoint circle/ellipse, lerp color gradient, punteros a buffer

Contexto · Escribí un programa que cree un dumb buffer de 800×600 a 32bpp e dibuje varias primitivas geométricas en él. El resultado debe verse como una escena compuesta con:

- Un fondo degradado horizontal (rojo → amarillo → verde → azul → morado).
- Una línea diagonal gruesa cruzando el canvas.
- Un círculo centrado de radio 150.
- Dos elipses superpuestas, una horizontal y otra vertical.

Consigna · El programa debe:

1. Crear un dumb buffer de 800×600 a 32bpp.
2. Dibujar un fondo degradado horizontal usando `lerp_color` (5 colores equidistantes).
3. Trazar una línea de Bresenham de `(50, 50)` a `(750, 550)` en blanco.
4. Dibujar un círculo centrado `(400, 300)` con radio `150` en rojo puro.
5. Dibujar una elipse centrada `(400, 300)` con rx=200, ry=100 en cian.
6. Dibujar otra elipse centrada `(400, 300)` con rx=100, ry=200 en magenta.

Criterios de aceptación

- [ ] Crea dumb buffer de 800×600 a 32bpp correctamente.
- [ ] Fondo degradado horizontal: 5 colores equidistantes usando `lerp` .
- [ ] Línea diagonal dibujada con Bresenham (sin huecos).
- [ ] Círculo centrado con radio 150 usando midpoint.
- [ ] Dos elipses superpuestas (una más ancha, otra más alta) con midpoint.
- [ ] Limpia: munmap → destroy en orden correcto.

Restricciones / pistas

- NO usar bibliotecas de gráficos (no SDL, no OpenGL, no Cairo).
- Usar solo `ioctl`, `mmap`, y matemática entera.
- Protección de boundaries: verificar que `(x, y)` estén dentro del buffer antes de escribir.

PARTE 8 — Framebuffers

Hasta ahora dibujaste píxeles en un dumb buffer... pero el CRTC no sabe que ese buffer existe. Necesitás crear un **framebuffer** asociado al handle del dumb buffer, para que el kernel sepa: "este buffer de VRAM es lo que hay que escanear".

8.1 `drmModeAddFB` — Asociar buffer a framebuffer

Un framebuffer (fb) en DRM es una estructura del kernel que dice "este buffer es una pantalla": tiene un handle, un width, height, pitch y un format (XRGB8888, ARGB8888, etc.).

```c
#include <drm_mode.h>

uint32_t fb_id;

// Crear framebuffer a partir del dumb buffer
int ret = drmModeAddFB(fd, mode->hdisplay, mode->vdisplay, 24, 32,
                       dbuf.pitch, dbuf.handle, &fb_id);
if (!ret) {
    printf("Framebuffer creado: fb_id=%u\n", fb_id);
} else {
    perror("drmModeAddFB falló");
}

// Liberar cuando ya no lo necesitás
drmModeRmFB(fd, fb_id);
```

Parámetros de `drmModeAddFB`:

- `width`, `height` — *dimensiones del framebuffer.*
- `depth` — *bits por píxel (24 para 32bpp con padding).*
- `bpp` — *bytes totales por píxel (32 para XRGB8888).*
- `pitch` — *bytes por fila (NO width * bytesPerPixel, sino el pitch real del dumb buffer).*
- `handle` — *handle del dumb buffer.*
- `&fb_id` — *salida: ID único del framebuffer.*

8.2 Activar el framebuffer con `drmModeSetCrtc`

Una vez que tenés el fb_id y un CRTC disponible, le decís al kernel "mostrá este framebuffer":

```
// Mostrar este framebuffer en el CRTC seleccionado
uint32_t connector_ids[16];  // máximo 16 conectores
int n_connectors = 1;
connector_ids[0] = <tu_connector_id>;

ret = drmModeSetCrtc(fd, crtc_id, fb_id, 0, 0,
                     connector_ids, n_connectors, mode);
if (ret) {
    perror("drmModeSetCrtc falló");
}
```

`drmModeSetCrtc` es el comando "haz todo" del modo legacy:

- *Asigna el framebuffer al CRTC.*
- *Posiciona el fb en el CRTC (x=0, y=0).*
- *Aplica el modo de resolución.*
- *Activa los conectores especificados.*

8.3 El flujo completo: dumb buffer → fb → show → draw → flip

El ciclo mínimo es:

```
// 1. Crear dumb buffer
create_dumb_buffer(fd, width, height, 32, &dbuf);

// 2. Crear framebuffer a partir del handle
drmModeAddFB(fd, width, height, 24, 32, dbuf.pitch, dbuf.handle, &fb_id);

// 3. Activar el framebuffer en el CRTC (esto "enciende" la pantalla)
drmModeSetCrtc(fd, crtc_id, fb_id, 0, 0, &connector_id, 1, mode);

// 4. Dibujar algo en el buffer mapeado
uint8_t *buf = (uint8_t *)dbuf.addr;
draw_rect(buf, pitch, width, height, color);

// 5. Limpiar y salir
drmModeRmFB(fd, fb_id);
munmap(dbuf.addr, dbuf.size);
destroy_dumb_buffer(fd, &dbuf);
```

> ***¡Importante!*** *En este punto, la pantalla se "enciende" después del* `SetCrtc`*. Hasta ahí, todo lo que dibujaste en el buffer era invisible porque el CRTC no lo estaba escaneando. Ahora sí aparece. Pero si queres animar, necesitás* ***page flip*** *(Parte 9).*

Práctica de la Parte 8

Ejercicio 25.7 — Encender una pantalla con color sólido

Conceptos · `drmModeAddFB`, `drmModeSetCrtc`, dumb buffer, framebuffer

Contexto · Escribí un programa que encienda tu monitor principal con un color sólido (azul). Esto demuestra el flujo mínimo: dumb buffer → framebuffer → SetCrtc → llenar pantalla.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0`, obtener recursos, activar atomic KMS.
2. Encontrar un conector conectado y su modo preferido.
3. Crear un dumb buffer del tamaño del modo a 32bpp.
4. Mapearlo a memoria CPU con `mmap()`.
5. Crear un framebuffer con `drmModeAddFB`.
6. Activarlo en el CRTC compatible con `drmModeSetCrtc`.
7. Llenar toda la pantalla con azul (0x0000FF en ARGB).
8. Esperar 3 segundos para que veas el color.
9. Limpiar: RmFB → munmap → destroy dumb buffer.

Entrada · Ninguna (detecta automáticamente el primer monitor conectado).

Salida · Pantalla azul durante ~3 segundos, luego vuelve a la normalidad.

Identificadores sugeridos `resources`, `connId`, `crtcId`, `mode`, `dbuf`, `fbId`

Criterios de aceptación

- ☐ Detecta el primer conector conectado y su modo preferido.
- ☐ Crea dumb buffer del tamaño del modo.
- ☐ Crea framebuffer con drmModeAddFB sin errores.
- ☐ Activa pantalla con drmModeSetCrtc (la pantalla cambia a azul).
- ☐ Espera 3 segundos (usando `sleep(3)` o `usleep`).
- ☐ Limpia correctamente en orden: RmFB → munmap → destroy.

Restricciones / pistas

- NO usar bibliotecas de ventanas (ni X11, ni Wayland, ni SDL).
- Si tu entorno usa un display manager que toma el control del DRM, podría bloquear SetCrtc. Ejecutá con `sudo` o en una TTY sin servidor gráfico (Ctrl+Alt+F3).

PARTE 9 — Page flip + vblank

En la Parte 8 viste cómo encender la pantalla con un color sólido. Pero si dibujás directamente en el buffer visible, vas a ver **tearing** (la mitad superior de una imagen y la mitad inferior de otra al mismo tiempo). Para evitarlo, necesitás **page flip sincronizado con vblank**.

9.1 ¿Qué es tearing?

Cuando el CRTC escanea píxeles del buffer y vos lo modificás al mismo tiempo:

```
Escaneo (CRTC) → ██████████░░░░░░ ← leyendo mitad superior
Dibujo         →           ████████ ← escribiendo mitad inferior
```

El resultado en pantalla es una imagen partida con dos fases distintas. Eso es **tearing**.

9.2 Double buffering: front buffer + back buffer

La solución estándar es **doble buffering**: tenés DOS buffers, uno visible (front) y otro donde dibujás (back):

```
Fase 1: Dibujar en back buffer
  Front (visible): [████████]   ← CRTC lo escanea
  Back (escondido):[░░░░░░░░]   ← vos dibujás aquí

Fase 2: Page flip al próximo vblank
  Front (visible): [░░░░░░░░]   ← se volcó el back
  Back (escondido):[████████]   ← listo para siguiente frame
```

El **page flip** es un ioctl que le dice al kernel "al próximo vblank, mostrá el otro buffer". Vblank = vertical blanking = el momento en que el CRTC termina de escanear una frame completa y está reiniciando desde arriba. Es el ÚNICO momento seguro para cambiar qué buffer se muestra.

9.3 Page flip con `drmModePageFlip`

```c
#include <drm.h>
#include <drm_mode.h>

// Parámetros opcionales de page flip (pueden ser NULL)
struct drm_mode_crtc_page_flip flip = {0};
flip.crtc_id      = crtc_id;
flip.fb_id        = new_fb_id;         // el buffer a mostrar al próximo vblank
flip.flags        = DRM_MODE_PAGE_FLIP_EVENT;  // quiere notificación cuando termine
flip.user_data    = your_callback_ptr; // datos pasados al handler

// Enviar page flip (no bloquea)
int ret = ioctl(fd, DRM_IOCTL_MODE_PAGE_FLIP, &flip);
if (ret < 0) {
    perror("DRM_IOCTL_MODE_PAGE_FLIP falló");
}
```

`DRM_MODE_PAGE_FLIP_EVENT`: *Si usás esta flag, el kernel envía un evento* `DRM_EVENT_FLIP_COMPLETE` *cuando el page flip se completa. Así sabés cuándo podés empezar a dibujar en el back buffer sin riesgo de que el CRTC lo esté escaneando.*

9.4 Escuchar eventos con `poll()` + handler

El fd del DRM es un file descriptor normal. Podés hacer `poll()` o `epoll()` sobre él:

```c
// Preparar poll para esperar eventos del DRM
struct pollfd pfd;
pfd.fd     = fd;
pfd.events = POLLIN | POLLOUT;  // queremos leer eventos

while (1) {
    int ret = poll(&pfd, 1, 16);  // timeout: 16ms (~60 fps max)
    if (ret < 0) {
        perror("poll falló");
        continue;
    }

    if (pfd.revents & (POLLERR | POLLHUP | POLLNVAL)) {
        fprintf(stderr, "Error en fd del DRM\n");
        break;
    }

    // Procesar eventos DRM
    drmHandleEvent evctx = {0};
    evctx.version = DRM_EVENT_CONTEXT_VERSION;
    evctx.page_flip_handler = page_flip_handler;  // tu callback
    evctx.sequence_handler = vblank_handler;       // opcional

    drmHandleEvent(&evctx);
}

// Callback de page flip: se llama cuando el flip terminó
void page_flip_handler(int fd, unsigned int frame,
                       unsigned int sec, unsigned int usec, void *data) {
    printf("Page flip completado en frame %u\n", frame);

    // ¡Acá podés empezar a dibujar en el back buffer!
    uint8_t *buf = (uint8_t *)flip_data->back_buffer_addr;
    draw_next_frame(buf, flip_data);
}
```

`drmHandleEvent`: *Es la función que interpreta los eventos crudos del fd DRM. Traduce events como `PAGE_FLIP`, `VBLANK`, `SEQ_UPDATE`, etc. en llamadas a tus callbacks.*

9.5 El ciclo de animación completo

El loop principal para una aplicación gráfica con KMS se ve así:

```
// Setup inicial (Parte 8): dumb buffer → fb → SetCrtc
setup_display(fd, resources);

// Inicializar double buffering
uint32_t fbs[2] = {fb0_id, fb1_id};
void   *bufs[2] = {addr0, addr1};  // ambos mapeados con mmap
int     front = 0;                  // cuál buffer es el visible ahora

while (running) {
    // 1. Esperar poll() → page_flip_handler
    //    (se ejecuta en el handler, no acá)

    // 2. Dibujar en el back buffer (el que NO está en front)
    int back = 1 - front;
    draw_frame(bufs[back], pitch, width, height);

    // 3. Enviar page flip al próximo vblank
    struct drm_mode_crtc_page_flip flip = {0};
    flip.crtc_id   = crtc_id;
    flip.fb_id     = fbs[back];
    flip.flags     = DRM_MODE_PAGE_FLIP_EVENT;
    flip.user_data = &front;

    ioctl(fd, DRM_IOCTL_MODE_PAGE_FLIP, &flip);

    // 4. front se vuelve el back para la próxima iteración
    front = back;
}
```

Importante: *el* `page_flip_handler` *se ejecuta desde dentro de* `drmHandleEvent()`*, que a su vez se llama desde tu loop principal (dentro de* `poll()`*). No es un thread separado. La ejecución del callback es síncrona respecto al* `drmHandleEvent()`*.*

Práctica de la Parte 9

Ejercicio 25.8 — Animación de rebote con page flip

Conceptos · Page flip, vblank events, poll(), double buffering

Contexto Escribí un programa que muestre una pelota blanca rebotando sobre un fondo azul. La pelota se mueve diagonalmente y rebotá en los bordes de la pantalla. Usá page flip sincronizado con vblank para evitar tearing.

Consigna · El programa debe:

1. Configurar el display (dumb buffer → fb → SetCrtc) como en el ejercicio 25.7.
2. Crear doble buffering: dos framebuffers, dos dumb buffers mapeados.
3. Dibujar una pelota blanca (círculo con radio ~30px) centrada en la pantalla.
4. Mover la pelota diagonalmente (~2 píxeles por frame).
5. Detectar colisiones con los bordes y rebotar (invertir dirección).
6. Usar page flip con `DRM_MODE_PAGE_FLIP_EVENT` al final de cada frame.
7. Ejecutar el loop hasta que se presione Ctrl+C.

Entrada · Ctrl+C para salir.

Salida · Pelota blanca rebotando suavemente sobre fondo azul, sin tearing.

Identificadores sugeridos `fbs`, `buffers`, `frontBuf`, `backBuf`, `ballX`, `ballY`, `velX`, `velY`

Criterios de aceptación

- [ ] Doble buffering con 2 dumb buffers + 2 framebuffers.
- [ ] Page flip al vblank (no hay tearing visible).
- [ ] Pelota se mueve diagonalmente y rebota en los 4 bordes.
- [ ] Usa `poll()` + `drmHandleEvent()` para esperar page flip completion.
- [ ] Limpia todos los resources al salir.

PARTE 10 — KMS atómico

El **modo legacy** (`drmModeSetCrtc`) funciona, pero tiene limitaciones: no podés configurar múltiples CRTCs simultáneamente, ni hacer changes atómicos (todos o nada). La **API atómica** resuelve eso.

10.1 ¿Por qué KMS atómico?

Con el modo legacy, cada llamada a `SetCrtc` reemplaza la configuración anterior. Si intentás configurar dos monitores con llamadas separadas, uno puede "robarte" el CRTC del otro.

La API atómica te permite configurar **todos los recursos en una sola operación atómica**: o todo cambia, o nada cambia. Es como un `commit` de base de datos para tu configuración gráfica.

10.2 Estructura básica de la API atómica

```c
#include <drm_fourcc.h>
#include <libdrm/drm_mode_atomic.h>

// 1. Crear un request atómico
uint32_t req_id;
int ret = drmIoctl(fd, DRM_IOCTL_MODE_ATOMIC_ALLOC, &req_id);
if (ret) { perror("Atomic alloc failed"); }

// 2. Obtener o crear un "state object" para cada recurso
drmModePropertyBlobProps *blob_props;
uint32_t modeset_blob_id;
drmModeGetPropertyBlob(fd, existing_blob_id, &blob_props, &modeset_blob_id);

// Para crear un blob nuevo (ej: configuración de CRTC):
struct {
    uint32_t crtc_id;
    uint32_t fb_id;
    uint32_t mode_blob_id;
    uint32_t connectors_ptr;  // pointer a array de connector IDs
    uint32_t count_connectors;
} modeset_state = {0};

// ... llenar modeset_state ...
uint32_t blob_len = sizeof(modeset_state);
void *blob_data = &modeset_state;

uint32_t new_blob_id;
drmModeCreatePropertyBlob(fd, blob_data, blob_len, &new_blob_id);

// 3. Agregar properties al request atómico
drmModeAtomicAddReqProp(req_id, crtc_id, PROP_MODE_ID, mode_blob_id);
drmModeAtomicAddReqProp(req_id, crtc_id, PROP_FB_ID, fb_id);
drmModeAtomicAddReqProp(req_id, connector_id, PROP_CRTC_ID, crtc_id);

// 4. Enviar el commit atómico
struct drm_mode_atomic atomic = {0};
atomic.req_ptr    = (uint64_t)req_id;
atomic.flags      = 0;  // DRM_MODE_ATOMIC_ALLOW_USERIO
atomic.reserved   = 0;  // padding obligatorio

ret = ioctl(fd, DRM_IOCTL_MODE_ATOMIC_COMMIT, &atomic);
if (ret) { perror("Atomic commit failed"); }

// 5. Liberar request
drmModeAtomicFree(req_id);
```

> ***`DRM_CLIENT_CAP_ATOMIC`:*** *Recordá que necesitás activar el modo atómico con* `drmSetClientCap` *(Parte 2.3) ANTES de usar cualquiera de estos ioctls. Si no lo activás, los ioctls atómicos fallarán con* `-EINVAL`.

10.3 Usando libdrm's atomic helpers (recomendado)

Hacer todo esto a mano es verboso. libdrm provee helpers que simplifican el proceso:

```c
#include <drm.h>
#include <libdrm/drm_mode_atomic.h>

// Crear contexto atómico
struct kms_device {
    int fd;
    struct kms_connector *connectors;
    struct kms_crtc *crtcs;
};

struct kms_state {
    struct drm_mode_atomic_req *req;
    // ... estado de cada CRTC, connector, fb ...
};

// Función para preparar un commit atómico:
int kms_commit(int fd, struct kms_state *state) {
    struct drm_mode_atomic atomic = {0};
    atomic.req_ptr = (uint64_t)state->req;
    atomic.flags = DRM_MODE_ATOMIC_NONBLOCK | DRM_MODE_ATOMIC_ALLOW_USERIO;

    int ret = ioctl(fd, DRM_IOCTL_MODE_ATOMIC_COMMIT, &atomic);
    return ret;
}
```

> ***NOTA:*** *Las funciones* `kms_*` *anteriores son pseudocódigo de ejemplo. La API real de libdrm para atómico es verbosa pero directa: creás blobs, agregás props al request, hacés commit. No hay una librería "KMS helpers" madura como SDL — esa es la gracia.*

10.4 Multi-monitor con KMS atómico

La gran ventaja del atómico es configurar múltiples CRTCs simultáneamente:

```
// Crear blobs para cada CRTC
struct {
    uint32_t crtc_id;
    uint32_t fb_id;
    uint32_t mode_blob_id;
} state0 = {.crtc_id = crtc_ids[0], .fb_id = fb_ids[0]};

struct {
    uint32_t crtc_id;
    uint32_t fb_id;
    uint32_t mode_blob_id;
} state1 = {.crtc_id = crtc_ids[1], .fb_id = fb_ids[1]};

uint32_t blob0, blob1;
drmModeCreatePropertyBlob(fd, &state0, sizeof(state0), &blob0);
drmModeCreatePropertyBlob(fd, &state1, sizeof(state1), &blob1);

// Agregar propiedades al request atómico
drmModeAtomicAddReqProp(req_id, crtc_ids[0], PROP_MODE_ID, mode_blob_ids[0]);
drmModeAtomicAddReqProp(req_id, crtc_ids[0], PROP_FB_ID, fb_ids[0]);
drmModeAtomicAddReqProp(req_id, connector_ids[0], PROP_CRTC_ID, crtc_ids[0]);

drmModeAtomicAddReqProp(req_id, crtc_ids[1], PROP_MODE_ID, mode_blob_ids[1]);
drmModeAtomicAddReqProp(req_id, crtc_ids[1], PROP_FB_ID, fb_ids[1]);
drmModeAtomicAddReqProp(req_id, connector_ids[1], PROP_CRTC_ID, crtc_ids[1]);

// Commit atómico: ambos CRTCs cambian simultáneamente
atomic.req_ptr = (uint64_t)req_id;
ret = ioctl(fd, DRM_IOCTL_MODE_ATOMIC_COMMIT, &atomic);
```

> ***Simultaneidad real:*** *Con el modo legacy, configurar un segundo monitor podría "robarle" el CRTC al primero. Con atómico, ambos CRTCs se configuran en el mismo kernel call. Si alguno falla, el commit entero es revertido.*

Práctica de la Parte 10

Ejercicio 25.9 — Dual monitor con KMS atómico

Conceptos · API atómica (DRM_IOCTL_MODE_ATOMIC_COMMIT), blobs, properties, multi-CRTC

Contexto · Escribí un programa que configure DOS monitores conectados simultáneamente usando la API atómica de KMS. Cada monitor debe tener su propia resolución y framebuffer.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0`, activar atomic KMS.
2. Enumerar recursos y encontrar al menos dos conectores conectados.
3. Para cada conector, elegir un CRTC compatible y un modo preferido.
4. Crear dumb buffers y framebuffers para cada monitor.
5. Usar `drmModeCreatePropertyBlob` para crear state blobs por CRTC.
6. Agregar todas las properties al request atómico (`DRM_IOCTL_MODE_ATOMIC_ALLOC`).
7. Hacer commit atómico con `DRM_IOCTL_MODE_ATOMIC_COMMIT`.
8. Dibujar un color sólido diferente en cada pantalla (azul + rojo).

Criterios de aceptación

- [ ] Detecta y configura DOS monitores conectados simultáneamente.
- [ ] Usa blobs de propiedades para cada CRTC.
- [ ] Agrega todas las properties al mismo request atómico.
- [ ] Commit atómico exitoso (ambas pantallas encendidas).
- [ ] Cada monitor muestra su color asignado (azul + rojo).

Restricciones / pistas

- Necesitás dos monitores físicamente conectados y un GPU con suficientes CRTCs.
- Si solo tenés un monitor, podés simular el segundo con un "fake connector" (pero no es ideal).
- En VM sin GPU passthrough, este ejercicio no va a funcionar.

PARTE 11 — Eventos DRM y hotplug

Hasta ahora configuraste pantallas estáticas. Pero en la vida real, los usuarios conectan y desconectan monitores todo el tiempo. DRM/KMS te avisa mediante **eventos del kernel**.

11.1 El evento de hotplug

Cuando conectás un cable HDMI o DisplayPort, el driver envía un evento `DRM_EVENT_HOTPLUG` al fd del DRM. Podés detectarlo con `poll()`:

```
void hotplug_handler(int fd, unsigned int frame,
                     unsigned int sec, unsigned int usec, void *data) {
    printf("Hotplug event: conectar/desconectar detectado\n");

    // Necesitás re-enumerar recursos porque algo cambió
    drmModeRes *resources = drmModeGetResources(fd);
    if (!resources) return;

    // Recorrer conectores y actualizar estados
    for (int i = 0; i < resources->countConnectors; i++) {
        uint32_t conn_id = resources->connector_id_ptr[i];
        drmModeConnector *conn = drmModeGetConnector(fd, conn_id);

        if (conn->connection == DRM_MODE_CONNECTED) {
            printf("Conector %u conectado\n", conn_id);
            // Configurar este connector...
        } else {
            printf("Conector %u desconectado\n", conn_id);
            // Desactivar CRTC de este connector...
        }

        drmModeFreeConnector(conn);
    }

    drmModeFreeResources(resources);
}
```

11.2 Registrar el handler de hotplug

En el loop principal, asociás tu callback al contexto de eventos:

```
drmHandleEvent evctx = {0};
evctx.version = DRM_EVENT_CONTEXT_VERSION;
evctx.page_flip_handler = page_flip_handler;
evctx.hotplug_handler = hotplug_handler;  // ← handler de hotplug
evctx.vblank_handler = vblank_handler;    // opcional

while (running) {
    struct pollfd pfd = {.fd = fd, .events = POLLIN};
    poll(&pfd, 1, 100);  // timeout: 100ms

    if (pfd.revents & POLLIN) {
        drmHandleEvent(&evctx);
    }
}
```

11.3 Tipos de eventos DRM

Evento	Cuándo se dispara	Handler
`DRM_EVENT_VBLANK`	Completó un vblank	`vblank_handler`
`DRM_EVENT_FLIP_COMPLETE`	Page flip terminado	`page_flip_handler`
`DRM_EVENT_CRTC_SEQUENCE`	Secuencia de CRTC completada	`crtc_sequence_handler`
`DRM_EVENT_HOTPLUG`	Conector conectado/desconectado	`hotplug_handler`
`DRM_EVENT_PAGE_FLIP_ERROR`	Page flip falló	`page_flip_handler` (con error)

Práctica de la Parte 11

Ejercicio 25.10 — Monitor de eventos DRM

Conceptos · poll(), drmHandleEvent, hotplug_handler, eventos DRM

Contexto · Escribí un programa que escuche eventos del fd DRM e imprima en consola cuando detecte cambios de estado (conexión/desconexión de monitores, page flips).

Consigna · El programa debe:

1. Abrir `/dev/dri/card0`, activar atomic KMS, configurar display básico.
2. Iniciar un loop con `poll()` en el fd del DRM.
3. Registrar handlers para: page_flip, hotplug y vblank (pueden ser stubs).
4. Al detectar un evento, imprimir qué tipo de evento fue y cuándo.
5. Mantenerse corriendo hasta Ctrl+C.

Criterios de aceptación

- [ ] Usa poll() en el fd del DRM con timeout razonable (~100ms).
- [ ] Registra hotplug_handler en drmHandleEvent.
- [ ] Imprime eventos detectados con timestamp.
- [ ] Limpia resources al salir (Ctrl+C).

PARTE 12 — OpenGL ES + EGL

Llegamos al punto donde la GPU hace el trabajo pesado. Hasta ahora dibujaste píxeles desde la CPU (matemática pura en C). Ahora vamos a usar **OpenGL ES** con **EGL** para renderizar directamente en un buffer de DRM, sin windowing system.

12.1 ¿Qué es EGL?

EGL es la "ponte" entre OpenGL ES y el sistema de composición de display de tu plataforma: X11, Wayland, Linux Framebuffer, o en nuestro caso... **DRM/KMS con GBM**.

Sin windowing system, necesitás:

- **libgbm** para crear buffers queOpenGL ES pueda usar.
- **EGL** para crear un contexto de OpenGL ES.
- El driver Mesa (o similar) para hacer el bridge entre EGL y DRM.

12.2 Inicialización EGL sobre GBM + DRM

```c
#include <GLES2/gl2.h>
#include <EGL/egl.h>
#include <gbm.h>
#include <drm_fourcc.h>

// Paso 1: Abrir el dispositivo DRM
int fd = open("/dev/dri/card0", O_RDWR | O_CLOEXEC);

// Paso 2: Crear un GBM surface (buffer para OpenGL)
struct gbm_device *gbm = gbm_create_device(fd);
if (!gbm) {
    perror("gbm_create_device falló");
}

// Paso 3: Crear un buffer GBM que EGL pueda usar
uint32_t width = 1920, height = 1080;
struct gbm_surface *gbm_surf = gbm_surface_create(
    gbm,
    width, height,
    GBM_FORMAT_XRGB8888,   // formato de píxel
    GBM_BO_USE_RENDERING | GBM_BO_USE_SCANOUT  // OpenGL + scanout
);

// Paso 4: Configurar EGL
EGLint attribs[] = {
    EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
    EGL_BLUE_SIZE,    1,
    EGL_GREEN_SIZE,   1,
    EGL_RED_SIZE,     1,
    EGL_NONE
};

EGLDisplay egl_dpy = eglGetDisplay((EGLNativeDisplayId)fd);
eglInitialize(egl_dpy, NULL, NULL);

EGLConfig egl_cfg;
EGLint num_configs;
eglChooseConfig(egl_dpy, attribs, &egl_cfg, 1, &num_configs);

// Paso 5: Crear contexto OpenGL ES
EGLint ctx_attribs[] = {
    EGL_CONTEXT_CLIENT_VERSION, 2,
    EGL_NONE
};
EGLContext egl_ctx = eglCreateContext(egl_dpy, egl_cfg,
                                      EGL_NO_CONTEXT, ctx_attribs);
```

```
// Paso 6: Crear surface EGL desde GBM surface
EGLSurface egl_surf = eglCreateWindowSurface(egl_dpy, egl_cfg,
                                             (EGLNativeWindowType)gbm_surf, NULL);

// Paso 7: Hacer el contexto current
eglMakeCurrent(egl_dpy, egl_surf, egl_surf, egl_ctx);
```

GBM_FORMAT_XRGB8888: *Es el formato de píxel estándar para OpenGL ES sobre DRM. Asegura que los shaders reciban los colores en el orden correcto (no invertido como a veces pasa con X11).*

12.3 Dibujar un triángulo con shaders

OpenGL ES es **baseless**: necesitás al menos un vertex shader y un fragment shader:

```
// Vertex shader simple: pasa coordenadas directas a clip space
const char *vertex_shader =
    "attribute vec4 pos;\n"
    "void main() {\n"
    "  gl_Position = pos;\n"
    "}\n";

// Fragment shader: color sólido (rojo)
const char *fragment_shader =
    "precision mediump float;\n"
    "void main() {\n"
    "  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n"  // Rojo puro
    "}\n";

// Compilar shaders...
GLuint create_shader(GLenum type, const char *src) {
    GLuint shader = glCreateShader(type);
    glShaderSource(shader, 1, &src, NULL);
    glCompileShader(shader);

    GLint ok;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &ok);
    if (!ok) {
        GLchar log[512];
        glGetShaderInfoLog(shader, 512, NULL, log);
        fprintf(stderr, "Shader compile failed: %s\n", log);
    }
    return shader;
}

// Crear programa
GLuint vs = create_shader(GL_VERTEX_SHADER, vertex_shader);
GLuint fs = create_shader(GL_FRAGMENT_SHADER, fragment_shader);

GLuint program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glBindAttribLocation(program, 0, "pos");  // location 0 = "pos" attribute
glLinkProgram(program);

GLint link_ok;
 glGetProgramiv(program, GL_LINK_STATUS, &link_ok);
if (!link_ok) {
    GLchar log[512];
    glGetProgramInfoLog(program, 512, NULL, log);
```

```
    fprintf(stderr, "Program link failed: %s\n", log);
}

glUseProgram(program);
```

12.4 Dibujar y presentar

```
// Limpiar pantalla con negro
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

// Dibujar triángulo: vertices en clip space [-1, +1]
GLfloat vertices[] = {
     0.0f,  0.5f, 0.0f,   // cima
    -0.5f, -0.5f, 0.0f,   // izquierda abajo
     0.5f, -0.5f, 0.0f    // derecha abajo
};

GLuint pos_loc = glGetAttribLocation(program, "pos");
glEnableVertexAttribArray(pos_loc);
glVertexAttribPointer(pos_loc, 3, GL_FLOAT, GL_FALSE,
                      3 * sizeof(float), vertices);

glDrawArrays(GL_TRIANGLES, 0, 3);

// Presentar al GBM buffer
eglSwapBuffers(egl_dpy, egl_surf);
```

`eglSwapBuffers`: *En EGL sobre DRM/GBM, este swap no hace lo que pensás. No cambia buffers visuales como en X11. Simplemente sincroniza el rendering de OpenGL con la superficie GBM. Para presentar al display, necesitás hacer un page flip manual con el buffer GBM actual.*

Presentación real a DRM desde EGL

```c
// Obtener el buffer GBM actual (el que OpenGL escribió)
struct gbm_bo *bo = gbm_surface_lock_front_buffer(gbm_surf);

// Crear framebuffer a partir del BO
uint32_t width, height;
gbm_bo_get_width(bo, &width);
gbm_bo_get_height(bo, &height);

uint32_t handles[4] = {0};
uint32_t strides[4] = {0};
uint32_t offsets[4] = {0};

// Map buffer para obtener handles de DRM
handles[0] = gbm_bo_handle_for_plane(bo, 0).u32;
strides[0] = gbm_bo_get_stride_for_plane(bo, 0);
offsets[0] = gbm_bo_get_offset(bo, 0);

uint32_t fb_id;
drmModeAddFB2(fd, width, height, GBM_FORMAT_XRGB8888,
              handles, strides, offsets, &fb_id, 0);

// Page flip para mostrar
drmModePageFlip(fd, crtc_id, fb_id, DRM_MODE_PAGE_FLIP_EVENT, NULL);

// Liberar el BO (pero NO el fb — page flip aún lo usa)
gbm_surface_release_buffer(gbm_surf, bo);

// Cuando llegue el page_flip_handler:
void page_flip_handler(int fd, unsigned int frame,
                       unsigned int sec, unsigned int usec, void *data) {
    // Acá sí podés borrar el fb que fue flippeado
    drmModeRmFB(fd, *(uint32_t *)data);
}
```

Cuidado con la sincronización: *Entre* `eglSwapBuffers` *y el page flip, hay un race condition potencial. OpenGL puede estar aún escribiendo cuando el CRTC empieza a leer. Para una app de producción real, necesitás fences (GL fence sync) para garantizar que OpenGL terminó antes del flip.*

12.5 Limpieza

```
eglMakeCurrent(egl_dpy, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
eglDestroySurface(egl_dpy, egl_surf);
eglDestroyContext(egl_dpy, egl_ctx);
eglTerminate(egl_dpy);

gbm_surface_destroy(gbm_surf);
gbm_device_destroy(gbm);

close(fd);
```

Práctica de la Parte 12

Ejercicio 25.11 — Triángulo colorido con OpenGL ES + EGL

Conceptos · EGL initialization, GBM surfaces, shaders GLSL, vertex attributes, eglSwapBuffers

Contexto · Escribí un programa que cree una ventana sin windowing system: abre la GPU via DRM, configura EGL, compila un par de shaders GLSL para dibujar un triángulo de color verde sobre fondo negro.

Consigna · El programa debe:

1. Abrir `/dev/dri/card0` y crear un dispositivo GBM.
2. Configurar EGL con el fd del DRM (eglGetDisplay + eglInitialize).
3. Crear una surface GBM de 800×600 a formato XRGB8888.
4. Crear contexto OpenGL ES 2.x y hacer current.
5. Compilar vertex shader + fragment shader que dibujen un triángulo verde.
6. Dibujar el triángulo con `glDrawArrays(GL_TRIANGLES, ...)`.
7. Presentar con `eglSwapBuffers` + page flip DRM manual.

Criterios de aceptación

- ☐ EGL initialized correctamente desde fd DRM.
- ☐ GBM surface creada y vinculada a EGL.
- ☐ Shaders GLSL compilados y enlazados sin errores.
- ☐ Triángulo verde visible (fondo negro, triángulo verde).
- ☐ Page flip manual presentado el buffer al display.

Restricciones / pistas

- Compilá con: `gcc prog.c -lGLESv2 -lEGL -ldrm -lgbm`.
- Si falla eglGetDisplay, intentá con `eglGetDisplay(EGL_DEFAULT_DISPLAY)`.
- Verificá que Mesa está instalada (`mesa-utils`, `libgl1-mesa-dri`).

PARTE 13 — Vulkan surfaceless

Vulkan es la sucesora moderna de OpenGL ES: más explícita, más controlada, pero también más verbosa. Con **surfaceless** (sin surface de windowing system), renderizás directamente a buffers GBM.

13.1 ¿Qué es "surfaceless" en Vulkan?

Normalmente Vulkan necesita un **swapchain** vinculado a una ventana (X11, Wayland, Win32). En modo surfaceless, el swapchain se crea sobre GBM buffers — no hay ventanas involucradas.

```
#include <vulkan/vulkan.h>
#include <gbm.h>

VkInstance instance;
VkPhysicalDevice phys_dev;
VkDevice device;
VkQueue graphics_queue;
VkSurfaceKHR surface;       // "surfaceless" = sin windowing system
VkSwapchainKHR swapchain;
```

13.2 Crear la Vulkan Instance

```
// Habilitar extensiones necesarias para DRM/GBM
const char *instance_exts[] = {
    VK_KHR_surface_EXTENSION_NAME,
    VK_KHR_display_EXTENSION_NAME,  // o VK_KHR_wayland_surface / VK_KHR_xlib_surface
};

VkApplicationInfo app = {0};
app.sType         = VK_STRUCTURE_TYPE_APPLICATION_INFO;
app.pApplicationName = "drm-kms-vulkan";
app.apiVersion    = VK_MAKE_VERSION(1, 1, 0);

VkInstanceCreateInfo create_info = {0};
create_info.sType           = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
create_info.pApplicationInfo = &app;
create_info.enabledExtensionNamecount = 2;
create_info.ppEnabledExtensionNames   = instance_exts;

vkCreateInstance(&create_info, NULL, &instance);
```

Extensiones: *Para DRM/KMS puro (sin X11/Wayland), necesitás* `VK_KHR_display` *o un loader específico como* `libvulkan-wayland` *o* `libvulkan-radeon`*. La forma más universal es usar el* ***Vulkan Display Protocol*** *(* `VK_DISPLAY_POWER_STATE` *).*

13.3 Crear un swapchain surfaceless (GBM-backed)

Este es el paso más complejo. Necesitás crear un swapchain cuyos buffers vengan de GBM:

```
// Paso A: Crear VkSurfaceKHR desde GBM
// (Usamos la extensión VK_GOI_mesa_platform o similar para "surfaceless")

VkWin32SurfaceCreateInfoKHR win32 = {0};  // dummy, no usamos Win32
win32.hwnd     = NULL;
win32.hinstance = NULL;
vkCreateWin32SurfaceKHR(instance, &win32, NULL, &surface);

// Paso B: Configurar swapchain con GBM buffers
uint32_t width  = 1920;
uint32_t height = 1080;
uint32_t image_count = 2;  // doble buffering

VkSurfaceCapabilitiesKHR caps;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(phys_dev, surface, &caps);

// Validar que el surface soporte al menos 2 imágenes
if (caps.minImageCount < 2) {
    fprintf(stderr, "Surface no soporta doble buffering\n");
}
image_count = caps.minImageCount;  // usar el mínimo soportado

VkSwapchainCreateInfoKHR sc_create = {0};
sc_create.sType            = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
sc_create.surface          = surface;
sc_create.minImageCount    = image_count;
sc_create.imageFormat      = VK_FORMAT_B8G8R8A8_SRGB;  // o A8B8G8R8
sc_create.imageColorSpace  = VK_COLOR_SPACE_SRGB_NONLINEAR_KHR;
sc_create.imageExtent      = (VkExtent2D){ width, height };
sc_create.imageArrayLayers = 1;
sc_create.imageUsage       = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
sc_create.preTransform     = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
sc_create.compositeAlpha   = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
sc_create.imageSharingMode = SHARING_MODE_EXCLUSIVE;
sc_create.presentMode      = VK_PRESENT_MODE_IMMEDIATE_KHR;  // vsync o no?
sc_create.clipped          = VK_TRUE;

VkResult ret = vkCreateSwapchainKHR(device, &sc_create, NULL, &swapchain);
```

> ***Present mode:*** `VK_PRESENT_MODE_IMMEDIATE_KHR` *es "tear-through" (sin vblank). Si querés sincronizar con vblank, necesitás* `VK_PRESENT_MODE_FIFO_KHR` *(el modo seguro con page flip).*

13.4 Dibujar un frame en Vulkan

Vulkan es muy explícito: necesitás crear una **command buffer**, registrar comandos de draw, y

enviarla a la queue gráfica:

```
// Paso 1: Obtener imagen del swapchain
uint32_t image_index;
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX,
                      image_available_semaphores[image_index],
                      NULL, &image_index);

// Paso 2: Esperar signal (semáforo de que la imagen está disponible)
VkWaitForFencesInfo wait_info = {0};
wait_info.sType            = VK_STRUCTURE_TYPE_WAIT_FOR_FENCES_INFO;
wait_info.fenceCount       = 1;
wait_info.pFences          = &render_fences[frame_number];
wait_info.timeout          = UINT64_MAX;
vkWaitForFences(device, 1, wait_info.pFences, VK_TRUE, UINT64_MAX);

// Paso 3: Resetear la command buffer
vkResetCommandBuffers(cmd_buffers[image_index], 0, &cmd_buf);

VkCommandBufferBeginInfo begin = {0};
begin.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vkBeginCommandBuffer(cmd_buf, &begin);

// Paso 4: Render pass — inicio de la zona de draw
VkRenderPassBeginInfo rp_begin = {0};
rp_begin.sType                 = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
rp_begin.renderPass            = render_pass;
rp_begin.framebuffer           = framebuffers[image_index];
rp_begin.renderArea            = (VkRect2D){ {0, 0}, {width, height} };
rp_begin.clearValueCount       = 1;
rp_begin.pClearValues          = &clear_value;

vkCmdBeginRenderPass(cmd_buf, &rp_begin, VK_SUBPASS_CONTENTS_INLINE);

// Paso 5: Bind pipeline + draw
vkCmdBindPipeline(cmd_buf, VK_PIPELINE_BIND_POINT_GRAPHICS, graphics_pipeline);
vkCmdDraw(cmd_buf, 3, 1, 0, 0);  // 3 vertices, 1 instancia

// Paso 6: Finalizar render pass y command buffer
vkCmdEndRenderPass(cmd_buf);
vkEndCommandBuffer(cmd_buf);

// Paso 7: Submit a la graphics queue
VkSubmitInfo submit = {0};
submit.sType                = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submit.waitSemaphoreCount   = 1;
submit.pWaitSemaphores      = &image_available_semaphores[image_index];
submit.pWaitDstStageMask    = &pipeline_stage;
submit.commandBufferCount   = 1;
```

```
    submit.pCommandBuffers      = &cmd_buf;
    submit.signalSemaphoreCount = 1;
    submit.pSignalSemaphores    = &render_complete_semaphores[image_index];

    vkQueueSubmit(graphics_queue, 1, &submit, render_fences[frame_number]);

    // Paso 8: Present
    VkPresentInfoKHR present = {0};
    present.sType               = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
    present.waitSemaphoreCount  = 1;
    present.pWaitSemaphores     = &render_complete_semaphores[image_index];
    present.swapchainCount      = 1;
    present.pSwapchains         = &swapchain;
    present.pImageIndices       = &image_index;

    vkQueuePresentKHR(graphics_queue, &present);
```

> ***Vulkan es verboso:*** *Notá que hay ~30 llamadas de Vulkan por frame. Cada una con structs específicos, semáforos, fences, pipelines... Es el precio del rendimiento: control total sobre la GPU, pero boilerplate masivo. OpenGL ES te daba 10 líneas para lo mismo.*

13.5 Limpieza

```
    vkDeviceWaitIdle(device);

    vkDestroySwapchainKHR(device, swapchain, NULL);
    vkDestroySurfaceKHR(instance, surface, NULL);
    vkDestroyDevice(device, NULL);
    vkDestroyInstance(instance, NULL);

    close(fd);
```

Práctica de la Parte 13

Ejercicio 25.12 — Cuadrado rojo con Vulkan surfaceless

Conceptos · Vulkan instance, physical device, swapchain GBM-backed, render pass, command buffers

Contexto Escribí un programa minimalista que dibuje un cuadrado rojo sobre fondo negro usando Vulkan en modo surfaceless. Sin shaders personalizados — usá el pipeline por defecto de rasterización.

Consigna · El programa debe:

1. Crear Vulkan instance con extensiones para surface display.
2. Seleccionar physical device y crear logical device con queue gráfica.
3. Configurar un swapchain GBM-backed (800×600, doble buffering).
4. Crear pipeline raster básico + render pass (clear color negro → draw rojo).
5. Dibujar un rectángulo cubriendo toda la pantalla con color rojo sólido.
6. Submit command buffer → present frame.

Criterios de aceptación

- ☐ Vulkan instance creada sin errores.
- ☐ Swapchain configurado sobre GBM buffers (sin windowing system).
- ☐ Render pass limpia a negro y dibuja un rectángulo rojo.
- ☐ Frame presentado exitosamente al display.

Restricciones / pistas

- Compilá con: `gcc prog_vulkan.c -lvulkan -ldrm -lgbm`.
- Si `VK_KHR_display` no está disponible, usá `VK_KHR_surface` con un surface dummy GBM.
- Podés usar glslangValidator (shader compiler de SPIR-V) o construir el pipeline sin shaders.

PARTE 14 — Ejercicio jefe: Pelota rebotando en N monitores

Llegamos al final del capítulo. Este ejercicio combina **todo** lo que aprendiste:

- DRM/KMS básico (Parte 2–5): abrir GPU, enumerar recursos, configurar modos.
- Dumb buffers + framebuffers (Parte 6–8): crear buffers, mapearlos, asociar a CRTCs.
- Page flip + vblank (Parte 9): animación fluida sin tearing.
- KMS atómico (Parte 10): multi-monitor con configuración simultánea.
- Hotplug events (Parte 11): detectar cambios de monitores conectados.
- OpenGL ES/EGL o Vulkan: renderizar primitivas sobre los buffers DRM.

14.1 El desafío completo

> ***Escribí un programa que muestre una pelota blanca rebotando simultáneamente en TODOS los monitores conectados a tu GPU**, usando DRM/KMS directo sin windowing system.*

Requisitos obligatorios:

1. **Multi-monitor automático:** Detectá todos los conectores `DRM_MODE_CONNECTED` y configurá cada uno con su modo preferido. Usá KMS atómico para activarlos simultáneamente.
2. **Animación en tiempo real:** La pelota debe moverse diagonalmente rebotando en los bordes de CADA monitor (no solo el principal). Si tenés dos monitores lado a lado, la pelota debe atravesar de uno a otro sin detenerse.
3. **Doble buffering por CRTC:** Cada monitor tiene su propio par front/back buffer. Usá page flip con `DRM_MODE_PAGE_FLIP_EVENT` para sincronizar con vblank.
4. **Gradiente arco-by-arco:** La pelota NO es blanca — está coloreada con un gradiente que varía según el arco de rebote. Cada vez que la pelota toca un borde, cambia de color siguiendo el espectro RGB: rojo → naranja → amarillo → verde → cian → azul → violeta → rojo...
5. **Física de rebote:** La pelota tiene velocidad `(vx, vy)` en píxeles/frame. Rebotá invirtiendo la componente correspondiente cuando choque con un borde del monitor actual. Usá `lerp` para suavizar la transición de color entre arcos.
6. **Manejo de hotplug:** Si un nuevo monitor se conecta o desconecta durante la ejecución:
   - Reconectar → configurar automáticamente con su modo preferido.
   - Desconectar → quitar de la configuración (sin crash).
7. **Sin bibliotecas de windowing:** NADA de X11, Wayland, SDL, GLFW, Raylib... Solo libdrm, libgbm, y matemática en C puro. OpenGL ES / Vulkan son opcionales (podés dibujar las pelotas usando Bresenham/midpoint sobre dumb buffers como hiciste en la Parte 7).

14.2 Estructura de datos sugerida

```c
#define MAX_MONITORS 8

typedef struct {
    uint32_t conn_id;
    uint32_t crtc_id;
    uint32_t fb_id;
    uint32_t fb_handle;       // handle del dumb buffer
    void     *buf_addr;       // puntero mmap'ed
    uint32_t pitch;           // bytes por fila
    uint32_t width, height;
    drmModeModeInfo mode;
} MonitorConfig;

typedef struct {
    int   x, y;               // posición en píxeles (coordenada global)
    int   vx, vy;             // velocidad en píxeles/frame
    float color_hue;          // 0.0–1.0 para ciclo de arco-by-arco
    int   bounce_count;       // cuántos bordes tocó
} Ball;

typedef struct {
    int      fd;              // fd del dispositivo DRM
    MonitorConfig monitors[MAX_MONITORS];
    int      num_monitors;
    Ball     balls[4];        // hasta 4 pelotas simultáneas
    int      running;         // flag para el loop principal
} AppContext;
```

14.3 Conversión de hue a RGB (para el gradiente arco-by-arco)

Necesitás convertir un valor de "tonalidad" (`hue` , 0–1) a color RGB usando `lerp` :

```c
// Convierte hue (0-1) a RGB usando lerp — ciclo completo del espectro
void hsv_to_rgb(float hue, uint8_t *r, uint8_t *g, uint8_t *b) {
    int h = (int)(hue * 6.0f);
    float f = hue * 6.0f - h;

    // 6 sectores del espectro: rojo → amarillo → verde → cian → azul → magenta → rojo
    uint8_t sectors[6][3] = {
        {255, 255,   0},  // 0:   amarillo (rojo + verde)
        {  0, 255,   0},  // 1:   verde
        {  0, 255, 255},  // 2:   cian
        {  0,   0, 255},  // 3:   azul
        {255,   0, 255},  // 4:   magenta
        {255, 255,   0},  // 5:   vuelve a amarillo → rojo
    };

    uint8_t *c1 = sectors[h % 6];
    uint8_t *c2 = sectors[(h + 1) % 6];

    *r = lerp(c1[0], c2[0], f);
    *g = lerp(c1[1], c2[1], f);
    *b = lerp(c1[2], c2[2], f);
}

// Convierte hue a color ARGB 32bpp (XRGB8888)
uint32_t get_ball_color(float hue) {
    uint8_t r, g, b;
    hsv_to_rgb(hue, &r, &g, &b);

    // XRGB8888 en little-endian: 0xFFRRGGBB
    return ((uint32_t)0xFF << 24) |
           ((uint32_t)r << 16) |
           ((uint32_t)g <<  8) |
           ((uint32_t)b);
}
```

Ciclo de arco: *Cada vez que la pelota toca un borde (`bounce_count++`), avanzás el hue:* `ball.color_hue = fmodf(ball.bounce_count / 7.0f, 1.0f)` *. Así el color completa un ciclo después de 7 rebotes (un arco completo del espectro).*

14.4 Lógica de movimiento y colisión

```
void update_ball(AppContext *app, Ball *ball, uint64_t dt_ms) {
    // Determinar en qué monitor está la pelota
    int monitor_idx = find_monitor_at_pos(app, ball->x, ball->y);

    if (monitor_idx < 0) return;  // pelota no asignada a ningún monitor

    MonitorConfig *mon = &app->monitors[monitor_idx];

    // Actualizar posición (velocidad en píxeles/frame)
    int steps_per_frame = 2;  // velocidad base
    ball->x += ball->vx * steps_per_frame;
    ball->y += ball->vy * steps_per_frame;

    // Colisión con bordes horizontales del monitor actual
    if (ball->x < mon->width / BALL_RADIUS) {
        ball->x = mon->width / BALL_RADIUS;
        ball->vx = abs(ball->vx);  // rebotar: invertir dirección horizontal
        on_bounce(app, ball);      // actualizar color del arco
    } else if (ball->x > mon->width - mon->width / BALL_RADIUS) {
        ball->x = mon->width - mon->width / BALL_RADIUS;
        ball->vx = -abs(ball->vx);
        on_bounce(app, ball);
    }

    // Colisión con bordes verticales del monitor actual
    if (ball->y < BALL_RADIUS) {
        ball->y = BALL_RADIUS;
        ball->vy = abs(ball->vy);
        on_bounce(app, ball);
    } else if (ball->y > mon->height - BALL_RADIUS) {
        ball->y = mon->height - BALL_RADIUS;
        ball->vy = -abs(ball->vy);
        on_bounce(app, ball);
    }

    // ¿La pelota cruzó al monitor vecino?
    int next_mon = check_cross_monitor(app, ball->x, ball->y);
    if (next_mon >= 0 && next_mon != monitor_idx) {
        // Transferir la pelota al nuevo monitor
        printf("Pelota cruzó a monitor %d\n", next_mon);
    }
}

void on_bounce(AppContext *app, Ball *ball) {
    ball->bounce_count++;
    // Avanzar el ciclo de color: cada 7 rebotes = ciclo completo
```

```
    ball->color_hue = fmodf((float)(ball->bounce_count % 64) / 64.0f, 1.0f);
}
```

Coordenadas globales: *Usá un sistema de coordenadas global donde* `x = 0, y = 0` *es la esquina superior izquierda del monitor más a la izquierda. Si tenés dos monitores de 1920×1080 lado a lado:*

- *Monitor 0: x=0–1919, y=0–1079*
- *Monitor 1: x=1920–3839, y=0–1079 La pelota se mueve en este espacio global. Cada frame, la renderizás separadamente en cada monitor según su offset local.*

14.5 Renderizado por monitor

```
void render_frame(AppContext *app) {
    for (int i = 0; i < app->num_monitors; i++) {
        MonitorConfig *mon = &app->monitors[i];

        // Fondo: color sólido por monitor (azul para el primero, rojo para el
segundo...)
        fill_rect(mon->buf_addr, mon->pitch, mon->width, mon->height,
                  get_monitor_bg_color(i));

        // Dibujar pelotas visibles en este monitor
        for (int b = 0; b < 4; b++) {
            Ball *ball = &app->balls[b];
            if (is_ball_visible_in_monitor(ball, i)) {
                // Calcular posición relativa a este monitor
                int rel_x = ball->x - get_monitor_offset_x(app, i);
                int rel_y = ball->y;

                uint32_t color = get_ball_color(ball->color_hue);

                // Dibujar círculo con midpoint
                draw_circle_anti_alias(mon->buf_addr, mon->pitch,
                                       rel_x, rel_y, BALL_RADIUS, color);
            }
        }

        // Page flip si hay un framebuffer nuevo listo
        // (ver Parte 9 para detalles de page flip)
    }
}
```

14.6 Criterios de aceptación finales

El programa debe cumplir TODOS estos criterios para considerarse completo:

Funcionalidad básica (obligatorio)

- [ ] Abre `/dev/dri/card0`, enumera recursos, detecta al menos un conector conectado.
- [ ] Configura cada monitor conectado con su modo preferido (KMS atómico).
- [ ] Crea dumb buffers y framebuffers por cada monitor.
- [ ] Enciende TODOS los monitores simultáneamente con `drmModeSetCrtc` o atomic commit.

Animación

- [ ] Pelota blanca (o de color) visible en pantalla, moviéndose diagonalmente.
- [ ] Rebota suavemente en los bordes de cada monitor individual.
- [ ] Si la pelota cruza entre monitores, aparece en el siguiente sin saltos.
- [ ] Doble buffering con page flip + vblank (sin tearing).

Gradiente arco-by-arco

- [ ] La pelota cambia de color cada vez que toca un borde.
- [ ] El ciclo de colores sigue el espectro: rojo → naranja → amarillo → verde → cian → azul → violeta.
- [ ] Usa `lerp` para interpolación suave entre colores (sin saltos bruscos).
- [ ] Al completar 7–64 rebotes, el color vuelve a rojo (ciclo completo).

Multi-monitor

- [ ] Funciona con 1 monitor (degradado simple) y con 2+ monitores.
- [ ] Si un monitor se desconecta, la pelota que estaba en él desaparece sin crash.
- [ ] Si un monitor nuevo se conecta, aparece la pelota en el nuevo espacio disponible.

Código limpio

- [ ] Estructura de datos clara (`MonitorConfig`, `Ball`, `AppContext`).
- [ ] Sin fugas de memoria: todos los buffers y framebuffers liberados al salir.
- [ ] Manejo correcto de errores (no crash si un ioctl falla).
- [ ] Código organizado en funciones pequeñas y reutilizables.

14.7 Compilación del ejercicio jefe

```
# Con dumb buffers + Bresenham/midpoint (solo C puro + libdrm)
gcc exercise_25_jefe.c -ldrm -o exercise_25_jefe

# Con OpenGL ES/EGL (si preferís shaders en vez de punteros crudos)
gcc exercise_25_jefe.c -lGLESv2 -lEGL -ldrm -lgbm -lm -o exercise_25_egl

# Con Vulkan surfaceless (para los valientes)
gcc exercise_25_jefe.c -lvulkan -ldrm -lgbm -lm -o exercise_25_vulkan
```

Ejecución: *Corré el programa desde una TTY (Ctrl+Alt+F3), sin servidor gráfico activo. Si X11/Wayland están corriendo, van a "robar" el control de DRM y tu programa va a fallar en* `drmModeSetCrtc` *con* `-EBUSY` *.*

```
sudo systemctl stop gdm   # o sddm, lightdm, etc.
Ctrl+Alt+F3               # ir a TTY limpia
sudo ./exercise_25_jefe
```

14.8 Lo que te llevás de este capítulo

Resumen rápido de lo que aprendiste (y las herramientas que ahora tenés):

- **DRM/KMS es el camino directo al hardware gráfico.** Abrís `/dev/dri/cardN`, hablás con el kernel, y controlás directamente qué resolución se muestra y qué buffer se escanea. Sin intermediarios.
- **El flujo mínimo es siempre el mismo:** abrir fd → enumerar recursos → configurar CRTC + connector → crear dumb buffer + framebuffer → dibujar → page flip. Todo lo demás es variación de esto.
- **Bresenham y midpoint son las bases de la gráficos por computadora.** Sin GPU, sin OpenGL, sin bibliotecas. Solo sumas, restas y decision tree — eso es una línea en pantalla.
- **Page flip + vblank = cero tearing.** Cambiás el buffer visible solo cuando el CRTC terminó de escanear la frame anterior. Es la base de toda animación fluida en Linux.
- **KMS atómico es el futuro.** Legacy funciona para un monitor, pero multi-monitor real requiere blobs + properties + commit atómico.
- **OpenGL ES y Vulkan pueden correr sobre DRM sin windowing system.** EGL puentea entre GLES y GBM; Vulkan surfaceless crea un swapchain sobre buffers GBM. Sin X11, sin Wayland — solo tu GPU.

En el siguiente capítulo, veremos... bueno, depende de qué quieras aprender después. Pero por ahora:

> ***Felicidades.*** *Acabás de escribir gráficos nativos en Linux sin ninguna biblioteca de windowing system. Eso significa que entendés más del hardware gráfico que el 99% de los desarrolladores de videojuegos y apps — porque la mayoría NI SIQUIERA SABE que DRM existe.*

¿Qué sigue?

Este capítulo cierra el recorrido por gráficos en Linux sin intermediarios. Los próximos capítulos podrían explorar:

- **Archivos de configuración udev:** Cómo hacer hotplug automático sin reiniciar el sistema.
- **DRM/KMS con V4L2:** Captura de video directamente desde la GPU.
- **GPU computing con OpenCL:** Usar DRM para compartir buffers entre render y compute.
- **Wayland compositor mínimo:** Ahora que conocés DRM, escribí tu propio compositor en ~300 líneas.

¡Tu elección! El capítulo sobre DRM/KMS está completo — las ideas quedan a partir de ahí.