Fuchsia Internals — Volume I

Graphics & Display Pipeline

Fuchsia Internals Series

***Abstract*—Fuchsia OS decomposes its graphics pipeline into cooperating user-space services communicating via FIDL typed channels with zero-copy shared VMOs. This reference covers the complete stack: Magma GPU driver framework, sysmem buffer negotiation, display coordinator, Flatland 2D compositor, Escher Vulkan renderer, frame scheduling, and the Zircon memory model. Targeted at hardware engineers familiar with RTL pipeline design and FPGA GPU implementation.**

***Index Terms*—Fuchsia OS, GPU driver, Vulkan, display pipeline, compositing, sysmem, Flatland, Magma, Zircon, FIDL, hardware compositor**

I. Introduction

FUCHSIA is Google's capability-based microkernel OS with a novel graphics stack designed for low latency, hardware compositing, and strict process isolation. Unlike Linux's monolithic DRM/KMS subsystem, Fuchsia decomposes graphics into cooperating user-space services: **Magma** (GPU driver framework), **sysmem** (shared buffer negotiation), **display coordinator** (hardware display abstraction), **Flatland/Scenic** (compositor), and **Escher** (Vulkan renderer). All inter-component communication uses **FIDL** (Fuchsia Interface Definition Language) over Zircon kernel channels.

For an EE/FPGA engineer familiar with VHDL pipeline stages, the analogy is strong: each service is a pipeline stage with well-defined handshake signals (FIDL calls) and shared memory (VMOs playing the role of ping-pong frame buffers). A Zircon *channel* is analogous to an AXI4-Stream interface. It is typed, ordered, and flow-controlled. A *VMO* is a DMA buffer descriptor whose physical backing is managed by the kernel. A *handle* is an unforgeable capability token, like a hardware bus grant that the firmware cannot forge.

FIDL in particular serves as the *bus fabric* of the Fuchsia graphics stack. Every cross-process graphics call (from "allocate this buffer" to "scan this image") is a FIDL method call serialized over a Zircon channel. FIDL is strongly typed (struct/table/union), versioned, and generates C++/Rust/Dart bindings. Wire encoding is a compact binary format; for the purpose of this tutorial, treat FIDL definitions as IDL for the AXI subordinate port of each graphics IP block.

Volume I of the fourteen-volume *Fuchsia Internals* series (I: Graphics & Display Pipeline; II: Build System & Toolchains; III: The Zircon Kernel; IV: Starnix; V: Graphics FIDL APIs in Practice; VI: The Component Framework; VII: Driver Development; VIII: Storage; IX: Power Management; X: Networking; XI: FIDL; XII: Software Delivery & OTA; XIII: Diagnostics & Observability; XIV: Security & Verified Boot). Compiled 2026 from public Fuchsia OS source and documentation at `fuchsia.dev` and `cs.opensource.google/fuchsia`.

Aspect	Linux DRM/KMS	Fuchsia
GPU driver host	Kernel module	User-space driver host
GPU IPC	ioctl syscalls	FIDL/Zircon channels
Buffer allocation	GEM/DMA-BUF	sysmem collections
Compositor	KMS/Wayland	Flatland (2D)
Display API	DRM planes	Display coordinator FIDL
Security model	DAC + seccomp	Capability handles
GPU fault scope	System-wide	Per-connection

The capability model is fundamental: every handle held by a process is an explicit, kernel-mediated permission. When a process crashes, its handles are closed by the kernel, and all kernel objects whose refcount drops to zero are atomically freed. The crash of one app cannot leave the compositor or display hardware in an inconsistent state. There are no dangling GPU mappings and no zombie display layers.

A. Zircon Kernel Graphics Primitives

The Zircon kernel provides the following objects used by the graphics stack:

- **VMO (Virtual Memory Object)**: the universal buffer descriptor. Properties: size (multiple of 4KB), optional physical contiguity, optional cache policy override, child/-parent relationship for sysmem hierarchy. Key syscalls: `zx_vmo_create()`, `zx_vmo_create_child()`, `zx_vmar_map()` / `unmap()`.
- **Channel**: a bi-directional, ordered message pipe. Messages consist of up to 64KB of data bytes plus up to 64 handle slots. FIDL messages are sent as channel messages. A channel endpoint is a handle; closing it sends an "epitaph" to the peer, enabling error propagation across process boundaries.
- **Event**: a 1-bit synchronization object. `zx_event_signal(ZX_EVENT_SIGNALED)` sets it; `zx_object_wait_one()` blocks on it. Used for Magma semaphores, Flatland acquire/release fences, and Vulkan external semaphores.
- **Eventpair**: a pair of linked events. When one peer is closed, the other is signaled with `ZX_EVENTPAIR_PEER_CLOSED`. Used for ViewCreationToken/ViewportCreationToken pairs in Flatland. When the parent's Viewport is destroyed, the child View detects this automatically.
- **BTI (Bus Transaction Initiator)**: represents a DMA engine's IOMMU context. `zx_bti_pin()` pins VMO pages into the IOMMU, returning physical addresses valid for DMA. Each platform device (GPU, display controller) has one BTI.
- **PMT (Pinned Memory Token)**: the result of `zx_bti_pin()`. Holds the IOMMU mapping alive. Physical addresses are stable while the PMT exists. `zx_pmt_unpin()` releases the IOMMU mapping and allows pages to be moved.

- **Interrupt**: wraps a hardware IRQ line. `zx_interrupt_wait()` blocks the IRQ thread; the kernel calls this when the hardware asserts the interrupt line. Used by the MSD's interrupt handler thread.
- **Port**: an async notification hub. Multiple waitable objects (events, channels, interrupts) can be attached to a port; `zx_port_wait()` blocks until any of them trigger. Used by Scenic's dispatcher to multiplex vsync events, client Present() calls, and input events on a single thread.

1) FIDL Wire Encoding: FIDL messages are encoded in a compact binary format:

- **Header**: 16 bytes, holding a 4-byte txid, 4-byte flags, and an 8-byte ordinal (the method identifier).
- **Primary object**: the top-level struct/table, inline in the message.
- **Out-of-line objects**: vectors and strings are referenced by (offset, count) pointers into out-of-line data appended after the primary object.
- **Handles**: transferred in the handle slots of the channel message (not inline in the byte buffer). Handle indices in the wire format are `0xFFFFFFFF` sentinels replaced at send time by the actual handle at that index.

For the graphics path, the most performance-sensitive FIDL calls are `Flatland/Present()` (called once per frame, carrying fence handles) and `Primary/ExecuteCommandBuffers()` (called for each render pass, carrying a vector of `CommandBuffer` structs). These calls typically transmit <200 bytes of FIDL data plus 2–4 handle slots, adding <5μs of IPC overhead per call on modern hardware.

2) Component Framework and Capabilities: Fuchsia's Component Framework (CF2) manages capability routing between components. Graphics capabilities are routed in component manifests (`.cml` files):

Listing 1. App component manifest (abridged .cml)

```
{
  use: [
    // Connect to Flatland compositor:
    { protocol: "fuchsia.ui.composition.Flatland" },
    // Connect to buffer allocator:
    { protocol: "fuchsia.ui.composition.Allocator" },
    // Connect to sysmem for buffer allocation:
    { protocol: "fuchsia.sysmem2.Allocator" },
    // Connect to GPU (via devfs):
    { directory: "dev-gpu",
      rights: ["r*"],
      path: "/dev/class/gpu" },
  ],
}
```

The Component Manager enforces that an app can only access the graphics protocols explicitly listed in its manifest. An app without `fuchsia.ui.composition.Flatland` in its `use` list cannot render to screen. There is no way to reach it through ambient authority or path traversal.

II. Architecture Overview

Figure 1 shows the complete vertical stack. Physical GPU silicon is owned exclusively by the *Magma System Driver (MSD)*, a privileged user-space driver host process. GPU client apps load the *Installable Client Driver (ICD)*, a vendor-supplied `.so` implementing Vulkan. The ICD talks to the MSD over Zircon channels. Shared memory is always negotiated through *sysmem*. The *display coordinator* owns the display hardware (CRTC/planes), and only Scenic/Flatland speaks to it.

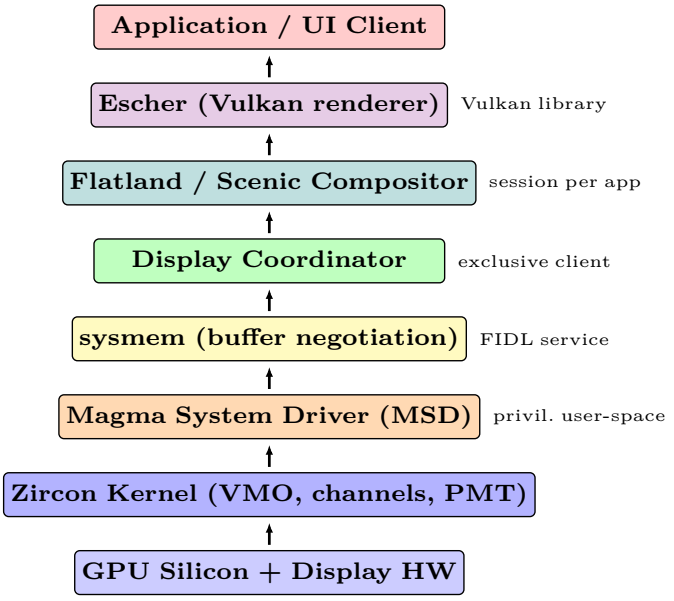


Fig. 1. Fuchsia full graphics stack.

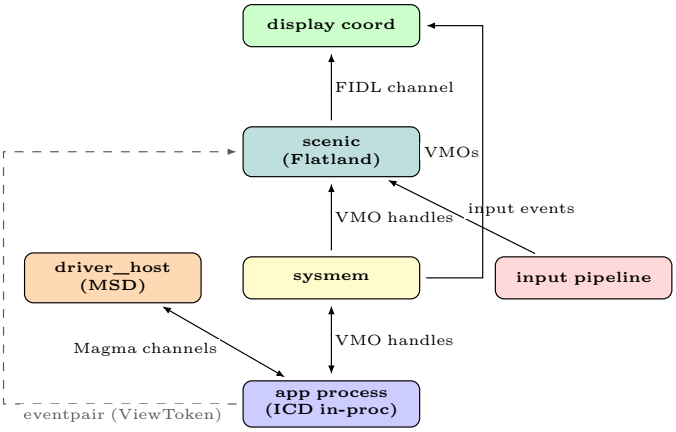


Fig. 2. Process topology: each box is a process; arrows are kernel object handles (channels/VMOs/events).

A. Process Isolation and Capability Model

Figure 2 shows the actual process topology. Each rounded box is a separate OS process; arrows are Zircon kernel objects (channels, VMOs, events, eventpairs) transferred as capability handles.

Key design axioms: (1) *No kernel GPU driver*: all GPU management is in privileged user-space. (2) *Zero-copy buffers*: sysmem negotiates formats so no pixel is ever copied between layers. (3) *Hardware compositing first*: Flatland sends layers directly to display planes when possible, bypassing the GPU entirely. (4) *Handle = capability*: a process has exactly the permissions represented by the handles it holds; no ambient authority.

III. Magma: GPU Driver Framework

A. Two-Component Model

Magma mirrors the Linux UMA DRM model but splits cleanly between user-space components:

- **MSD** (Magma System Driver): runs in a dedicated *driver host* process managed by the *Fuchsia Driver Framework (FDF)*.

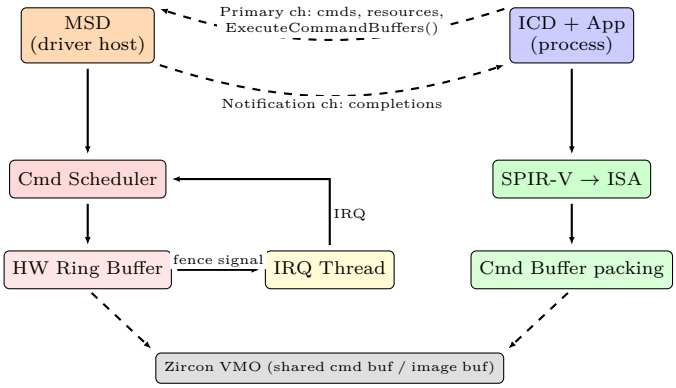


Fig. 3. Magma architecture: MSD in driver host, ICD in app, dual channels.

It exclusively programs GPU registers, manages interrupt threads, schedules command buffers, and owns the GPU hardware address space setup.

- **ICD** (Installable Client Driver): a vendor `.so` loaded into the *application* process. It compiles SPIR-V → ISA, builds command buffers, and sends them to the MSD via Zircon channels.

The split mirrors PCIe device ownership: MSD ≈ physical register map owner; ICD ≈ command generator running on the host side.

B. Connection Protocol

When an app calls `vkCreateDevice`, the ICD invokes `IcdLoaderDevice/GetIcdList()` to locate the vendor ICD, then calls `Device/Connect2()` on the MSD. This returns two Zircon channel handles:

1) **Primary channel**: carries resource creation (`ImportBuffer`, `ImportSemaphore`), command execution (`ExecuteCommandBuffers`), context management.
2) **Notification channel**: MSD-initiated vendor messages signaling completion events, used to implement Vulkan fences/semaphores.

Each MSD connection spawns a dedicated thread for that client, preventing one client's stall from blocking others.

C. Memory: VMO-Backed Buffers

GPU buffers are Zircon VMOs. The ICD: (1) allocates a VMO (`zx_vmo_create` or via sysmem); (2) maps it into CPU address space (`zx_vmar_map`) to write commands; (3) calls `ImportBuffer(vmo_handle)` on the Primary channel; MSD maps the same VMO into GPU address space; (4) submits `ExecuteCommandBuffers()` referencing the buffer handle; (5) the MSD programs the GPU ring/command buffer register to DMA from that VMO.

Two address-space management strategies exist: **client-managed** (ICD tracks GPU VAs; efficient but risk of GPU fault if unmap races command completion) and **system-managed** (MSD owns mappings; safer, but requires command buffer patching at submit time to fix GPU VA references).

D. Synchronization Primitives

Magma semaphores wrap `zx::event` objects. Both ICD and MSD can signal/wait on the same event handle duplicated across the connection. This maps directly to Vulkan `VkSemaphore` and `VkFence`: the ICD passes semaphore events as `wait_semaphores/signal_semaphores` in `ExecuteCommandBuffers()`, and the MSD programs hardware semaphore waits/signals into the command stream.

E. Scheduling and Error Isolation

The MSD implements configurable scheduling: FIFO, priority-based, or time-sliced with hardware preemption. GPU faults terminate only the offending connection; system-critical clients (e.g., the Scenic compositor) continue uninterrupted. The ICD receives a `ZX_ERR_IO` epitaph on its Primary channel and reports `VK_ERROR_DEVICE_LOST` to the application.

F. FIDL Snapshot

Listing 2. fuchsia.gpu.magma (abridged)

```
@transport("Driver")
closed protocol Device {
  strict GetIcdList() -> (struct {
    icd_list vector<IcdInfo>:MAX;
  });
  strict Connect2(resource struct {
    client_id    uint64;
    primary      server_end:Primary;
    notification client_end:Notification;
  }) -> ();
};
closed protocol Primary {
  strict ImportBuffer(resource struct {
    buffer zx.Handle:VMO;
  });
  strict ExecuteCommandBuffers(struct {
    command_buffers vector<CommandBuffer>:MAX;
  });
  strict MapBuffer(struct {
    hw_va      uint64;
    buffer_id  uint64;
    offset     uint64;
    length     uint64;
    map_flags  MapFlags;
  });
  strict ImportSemaphore(resource struct {
    semaphore zx.Handle:EVENT;
    id uint64;
  });
  strict ReleaseSemaphore(struct { id uint64; });
};
```

G. GPU Virtual Address Space Management

Each Magma connection owns an isolated GPU virtual address space (GPUVA). The MSD allocates a GPUVA region per connection at `Connect2()` time:

- On **ARM Mali**: each connection is assigned a per-context *ASID* (Address Space ID) written to the Mali MMU register. The hardware TLB is tagged with ASID so address spaces coexist without TLB flushes between clients.
- On **Intel GPU**: the equivalent is a per-process Graphics Translation Table (PPGTT). The MSD programs the per-context GEN_PDL register to point to the client's page directory.
- On **AMD GPU**: each client gets a per-VM page table set mapped via the VMID register in the CP.

The ICD requests mappings by sending `MapBuffer()` on the Primary channel, specifying the GPU virtual address, buffer handle, byte range, and `MapFlags`:

MapFlag	Meaning
`READ`	GPU read-only mapping
`WRITE`	GPU write enable
`EXECUTE`	GPU instruction fetch allowed
`WRITE_COMBINE`	Uncached writes (command ring)

A GPU page fault in one connection sends an interrupt to the MSD's IRQ thread. The MSD terminates that connection's

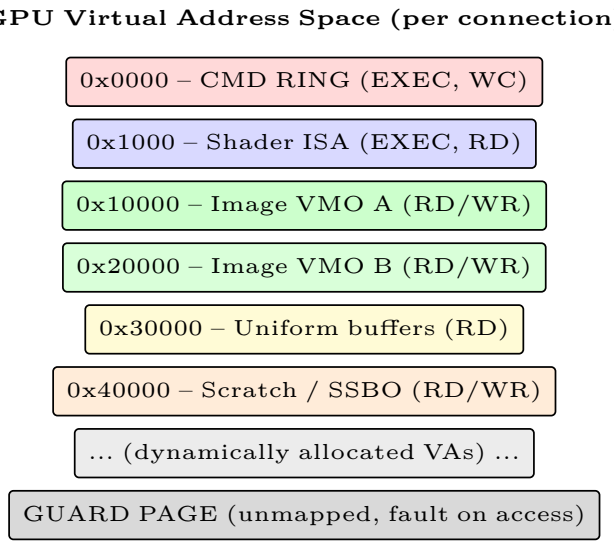


Fig. 4. GPU virtual address space layout per Magma connection.

GPUVA (unmaps all pages, invalidates the TLB for that ASID), destroys the hardware context, and sends `ZX_ERR_IO` on the Primary channel. Other connections' address spaces are unaffected.

H. Context and Submission Model

A Magma *context* corresponds to a GPU hardware context (register set saved/restored on preemption). The ICD calls `CreateContext()` on the Primary channel to obtain a `context_id`; Vulkan queues map 1:1 to Magma contexts.

`ExecuteCommandBuffers()` submits a vector of `CommandBuffer` structs:

Listing 3. CommandBuffer structure (C++ / Magma wire format)

```
struct CommandBuffer {
  uint64_t resource_id;   // VMO id containing cmds
  uint64_t start_offset;  // byte offset into VMO
  uint64_t start_size;    // byte length of cmd data
  // Resources to make GPU-resident before exec:
  vector<uint64_t> exec_resource_ids;
  // Semaphores to wait before starting:
  vector<uint64_t> wait_semaphore_ids;
  // Semaphores to signal after GPU completes:
  vector<uint64_t> signal_semaphore_ids;
  uint64_t context_id;    // target GPU context
};
```

The MSD resolves the semaphore wait list into hardware dependency chains. For Mali this inserts a *wait-for-dependency* atom into the job chain; for Intel this programs a MI_SEMAPHORE_WAIT command into the ring buffer before the batch start address.

I. Performance Counters

The `fuchsia.gpu.magma/PerformanceCounters` FIDL protocol allows userspace profiling tools to sample GPU hardware counters without kernel involvement:

Listing 4. PerformanceCounters protocol (abridged)

```
closed protocol PerformanceCounters {
  strict EnablePerformanceCounterAccess(resource struct {
    access_token zx.Handle:EVENT;
  }) -> ();
  strict CreatePerformanceCounterBufferPool(struct {
    pool_id uint64;
    event   zx.Handle:EVENT;
  }) -> ();
  strict AddPerformanceCounterBufferToPool(struct {
    pool_id   uint64;
    buffer_id uint64;
    buffer_offset uint64;
    buffer_size   uint64;
  }) -> ();
  strict TriggerPerformanceCounterReadIntoBuffer(struct {
    pool_id   uint64;
    trigger_id uint32;
  }) -> ();
  strict ClearPerformanceCounters(struct {
    counters vector<uint64>:MAX;
  }) -> ();
};
```

GPU counters (shader invocations, cache hit rates, memory bandwidth) are DMA'd directly into VMO-backed pools at each trigger. Tools like `fx trace` use this to build GPU flame graphs without any kernel patching.

J. Error Recovery

The MSD runs a *watchdog timer* (typically 2s) that fires if the GPU ring buffer HEAD register does not advance. On timeout:

1) MSD marks the active context as faulted.
2) GPU soft-reset: write reset register, wait for idle, re-initialize ring buffer.
3) All VMO pages mapped for that context are unmapped; page tables are freed.
4) `ZX_ERR_IO` is sent as an epitaph on the Primary channel; the connection is closed.
5) The Notification channel peer sees the epitaph; ICD reports `VK_ERROR_DEVICE_LOST`.

Scenic holds a *protected* GPU context with elevated priority. When a GPU reset occurs, the MSD restores Scenic's GPU context first (before any other context) to minimize compositor jank. Scenic detects the `VK_ERROR_DEVICE_LOST` event, recreates its Vulkan logical device, and re-imports all sysmem collections. This typically completes within one vsync period.

K. Power Management and Clock Gating

Magma integrates with Fuchsia's **SAG (System Activity Governor)** to manage GPU power domains:

1) When the first `ExecuteCommandBuffers()` arrives on an idle GPU, the MSD asserts a `fuchsia.power.broker/LeaseControl` on the GPU power domain. The power broker enables the GPU voltage rail and clock tree.
2) While any command buffer is pending, the lease is held. The GPU runs at a frequency selected by the MSD's DVFS (Dynamic Voltage and Frequency Scaling) policy based on queue depth.
3) After 50ms of GPU idle (no pending command buffers, notification channel quiet), the MSD releases the power lease. SAG enables GPU clock gating.
4) For sustained workloads (video playback, games), the MSD requests a "sustained" power level that disables DVFS transitions during the frame render window to avoid latency spikes from PLL re-lock delays.

On ARM Mali, clock gating is implemented via the `GPU_CONTROL.SHADER_PRESENT` and `TILER_POWER_TRANSIENT` registers. The MSD writes these registers directly after releasing the power lease. On Intel GPU, clock gating is managed by the hardware RC6 (Render C6) idle state, triggered by writing `0` to `GEN6_PMINTRMSK`.

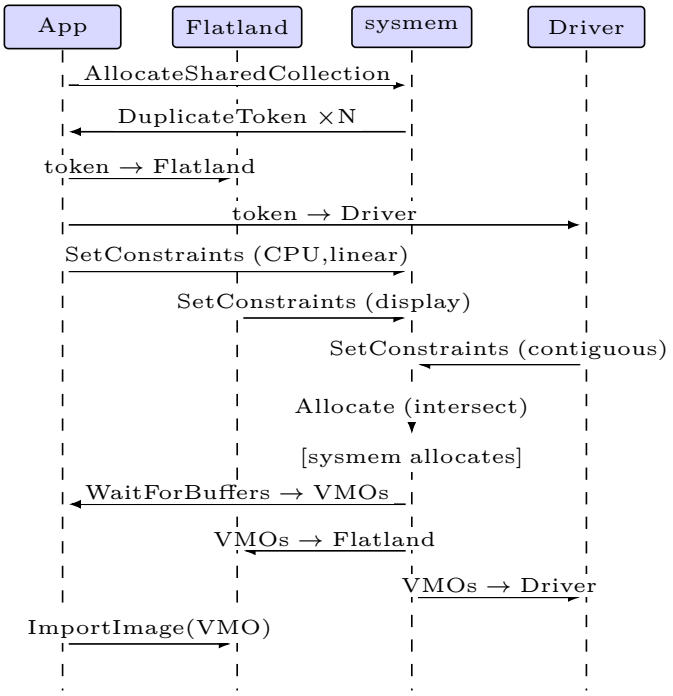


Fig. 5. sysmem buffer collection negotiation sequence.

L. Multi-Process GPU Sharing

Multiple processes can hold simultaneous Magma connections to the same MSD. The MSD serializes command buffer execution across connections:

- **Round-robin scheduling** (default): each connection's pending command buffers are interleaved at quantum boundaries. Quantum size is configurable (default: 2ms of GPU time).
- **Priority-based scheduling**: Scenic's GPU context runs at priority 15 (highest); application contexts at priority 8. High-priority contexts preempt lower-priority ones if the GPU supports hardware preemption (Mali Bifrost v7+, Intel Gen11+).
- **Temporal fairness**: a connection that submits many small command buffers does not starve connections that submit few large ones; the scheduler tracks GPU time consumed per connection across a 100ms sliding window.

IV. sysmem: Shared Buffer Negotiation

A. Purpose and Problem

When a GPU (ICD), display hardware (display coordinator), and a video decoder all need to share the same pixel buffer, they have conflicting requirements: the GPU wants tiled memory layout; the display expects linear scanout; the secure video decoder requires physically contiguous CPU-inaccessible memory. sysmem is the arbitration service that finds a format satisfying all participants *before* allocating, which eliminates copies.

B. Negotiation Protocol

The steps: (1) Initiator calls `AllocatorV2/AllocateSharedCollection()` to get a root token. (2) The token is duplicated, one copy per participant. (3) Each participant binds its token and calls `SetConstraints()`, declaring pixel formats, heap types, alignment, and minimum count. (4) sysmem intersects all constraints and calls the heap allocator. (5) All participants call `WaitForAllBuffersAllocated()` to get the VMOs and negotiated image format.

C. Constraints

A participant's `BufferCollectionConstraints` specifies:

- **Pixel formats**: ordered list of acceptable formats (e.g., `NV12`, `BGRA32`, `R8G8B8A8`).
- **Color spaces**: `sRGB`, `REC709`, etc.
- **Heap types**: `SYSTEM_RAM`, `AMLOGIC_SECURE` (DRM protected), `GOLDFISH_DEVICE_LOCAL` (emulator).
- **Memory flags**: `is_physically_contiguous` (required by some display planes), CPU accessibility, cache policy.
- **Size and pitch**: minimum width/height; minimum `bytes_per_row` (alignment for GPU texture fetch).

D. VMO Hierarchy

sysmem maintains a three-level VMO tree:

1) **Heap VMO**: allocated at boot from physical pool; never given to clients.
2) **Middle VMO**: child of Heap VMO, one per buffer; managed internally.
3) **Leaf VMO**: child of Middle VMO, distributed to participants. Clients name them via `SetName()` for debugging.

When all leaf VMO handles and mappings are released, the kernel fires `ZX_VMO_ZERO_CHILDREN` on the middle VMO; sysmem then returns that slot to the heap.

E. Late Joining and Dispensable Tokens

AttachToken allows new participants to join an *already-allocated* collection; the collection is reallocated if the new constraints conflict. **Dispensable tokens** allow a participant to fail constraint satisfaction without failing the whole collection. That is useful for optional hardware acceleration paths.

F. Pixel Format Intersection Algorithm

sysmem intersects participants' format lists by taking formats common to *all* participants' lists, preserving the priority order of the initiator's list. If the intersection is empty, `WaitForAllBuffersAllocated()` returns `ZX_ERR_NOT_SUPPORTED`.

Participant	Format list (priority order)
App (CPU write)	[BGRA32, NV12]
Display HW	[NV12, YUY2]
GPU ICD	[BGRA32, NV12, R8G8B8A8]
Intersection	[NV12] (only common format)

The intersection is computed per *format + modifier* pair: a format is only accepted if all participants agree on the same DRM modifier (e.g., linear, tiled, compressed).

G. Heap Allocator Internals

`SysmemContiguousPool` carves a physically contiguous region at boot from the bootloader-reserved memory range (passed via ZBI). It manages a buddy-allocator over this carve-out. `SysmemRamMemfd` delegates to Zircon's `zx_vmo_create()` for general system RAM. For protected heaps, the TrustZone secure monitor (`tee_client_api`) allocates and locks the physical range; the pages are programmed into the TZC-400 access-control registers before sysmem returns the VMO handle.

H. Buffer Naming and Debugging

Listing 5. sysmem debugging FIDL

```
closed protocol BufferCollection {
  // Named for fx sysmem dump output:
  strict SetName(struct {
    priority uint32;
    name string:64;
  }) -> ();
  // Identify the owning component:
  strict SetDebugClientInfo(struct {
    name string:64;
    id   uint64;
  }) -> ();
  // Purgeable: kernel may zero pages under memory pressure
  strict SetConstraintsAuxBuffers(struct {
    constraints AuxBufferConstraints;
  }) -> ();
};
```

Buffer names and client IDs appear in `fx sysmem dump` and in Zircon's memory pressure dumps, enabling identification of which component holds large buffer pools.

I. sysmem2 vs. sysmem v1

The sysmem2 API (`fuchsia.sysmem2`) replaces the original with table-based FIDL structs (versioned, extensible). Key changes: `BufferCollectionConstraints` becomes a table (unknown fields ignored); `PixelFormat` gains a `format_modifier` field for tiling (DRM modifiers like `I915_FORMAT_MOD_X_TILED`); `BufferMemoryConstraints` adds `heap_permitted` as a vector. Client code must check the platform sysmem API version via `Allocator/GetVmoInfo()`.

Listing 6. sysmem2 BufferCollectionConstraints (abridged)

```
type BufferCollectionConstraints = table {
  // Minimum number of buffers for this participant:
  1: min_buffer_count_for_camping uint32;
  2: min_buffer_count_for_shared_slack uint32;
  3: min_buffer_count uint32;
  4: max_buffer_count uint32;
  // Per-buffer memory constraints:
  5: buffer_memory_constraints BufferMemoryConstraints;
  // Per-image format constraints:
  6: image_format_constraints
       vector<ImageFormatConstraints>:MAX;
};
type BufferMemoryConstraints = table {
  1: min_size_bytes uint64;
  2: max_size_bytes uint64;
  3: physically_contiguous_required bool;
  4: secure_required bool;
  5: ram_domain_supported bool;
  6: cpu_domain_supported bool;
  7: inaccessible_domain_supported bool;
  8: heap_permitted vector<Heap>:MAX;
};
type ImageFormatConstraints = table {
  1: pixel_format PixelFormat;
  2: pixel_format_modifier uint64; // DRM modifier
  3: color_spaces vector<ColorSpace>:MAX;
  4: min_coded_width  uint32;
  5: max_coded_width  uint32;
  6: min_coded_height uint32;
  7: max_coded_height uint32;
  8: min_bytes_per_row uint32;
  9: bytes_per_row_divisor uint32;
};
```

V. Display Coordinator

A. Hardware Model

The display coordinator abstracts this hardware pipeline. Planes differ in hardware capability: **Primary** (full-screen, scaling, rotation, alpha blending); **Overlay** (chroma-key transparency, video under UI); **Sprite** (sub-frame rectangles); **Cursor** (64×64px, zero-latency position updates without reconfiguration).

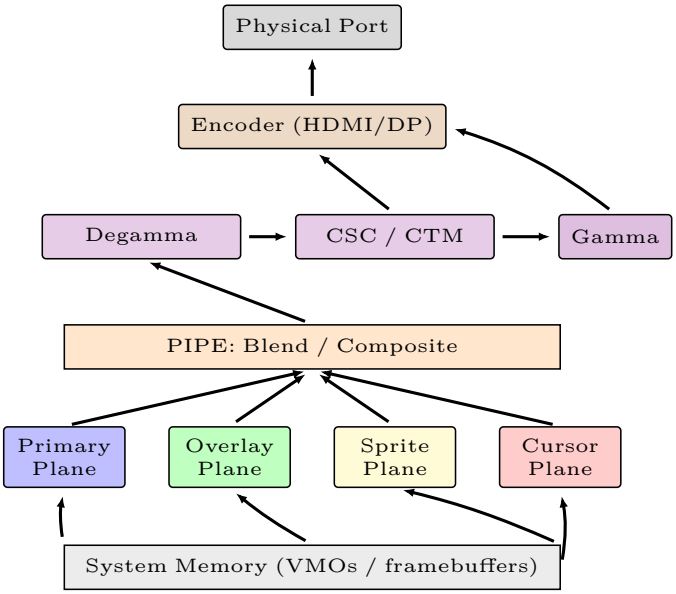


Fig. 6. Display controller hardware model: planes, pipe, color, encoder.

B. FIDL Protocol

Only one client may connect to the display coordinator at a time; Scenic holds this connection exclusively.

Listing 7. Display Coordinator FIDL (abridged)

```
closed protocol Coordinator {
  strict ImportImage(struct {
    image_config  ImageConfig;
    collection_id BufferCollectionId;
    index         uint32;
  }) -> (struct { image_id ImageId; }) error zx.Status;
  strict ReleaseImage(struct { image_id ImageId; });
  strict CreateLayer() -> (struct { layer_id LayerId; });
  strict SetLayerImage(struct {
    layer_id        LayerId;
    image_id        ImageId;
    wait_event_id   EventId;   // GPU release fence
    signal_event_id EventId;   // display retire signal
  });
  strict SetLayerPrimaryPosition(struct {
    layer_id   LayerId;
    transform  Transform;
    src_frame  Frame;  // source crop
    dest_frame Frame;  // destination on screen
  });
  strict CheckConfig(struct { discard bool; })
      -> (struct {
    res ConfigResult;
    ops vector<ClientCompositionOp>:MAX;
  });
  strict ApplyConfig();
  strict EnableVsync(struct { enabled bool; });
  strict SetDisplayColorConversion(struct {
    display_id     DisplayId;
    preoffsets     array<float32,3>;
    coefficients   array<float32,9>;
    postoffsets    array<float32,3>;
  });
  strict SetDisplayGammaTable(struct {
    display_id DisplayId;
    degamma    GammaTable;
    gamma      GammaTable;
  });
};
open protocol CoordinatorListener {
  strict OnVsync(struct {
    display_id     DisplayId;
    timestamp      zx.Time;
    applied_config ConfigStamp;
  });
  strict OnDisplaysChanged(struct {
    added   vector<Info>:MAX;
    removed vector<DisplayId>:MAX;
  });
};
```

C. Configuration Validation

Before `ApplyConfig()`, Scenic calls `CheckConfig()` to verify the proposed layer stack is achievable in hardware. The coordinator returns `OK` or a list of `ClientCompositionOp` hints indicating which layers must be GPU-composited before submission. This triggers Scenic's GPU compositing fallback path.

D. Vsync and Stamping

`OnVsync()` fires at the beginning of vertical blanking and reports the `ConfigStamp` of the configuration currently scanned out. Scenic uses this to: (a) compute actual presentation time, (b) release acquire fences, (c) feed the FrameScheduler latency model.

E. Display Timing and Vsync Generation

The display coordinator manages precise vsync timing. On HDMI/DisplayPort, the timing is defined by the active pixel area, front/back porches, and sync pulse widths encoded in the EDID `Detailed Timing Descriptor`. The coordinator programs these values into the display controller's timing generator registers (e.g., Intel's `HTOTAL`, `HBLANK`, `HSYNC`, `VTOTAL` registers).

The vsync interrupt fires at the beginning of the vertical blanking interval. The kernel's interrupt handler increments a monotonic vsync counter and signals the Zircon interrupt object held by the coordinator driver. The driver computes the timestamp using `zx_clock_get_monotonic()` at interrupt entry (before any scheduling latency), ensuring $<5\mu$s timestamp accuracy. This timestamp is reported to Scenic via `OnVsync()`.

For variable refresh rate (VRR/FreeSync/G-Sync) panels, the coordinator can adjust the vsync period dynamically. The `SetMinimumRgb()` and `SetDisplayMode()` FIDL calls support mode switching; VRR activation is gated on the display's EDID exposing the appropriate VESA VRR or AMD FreeSync capability block.

F. Multi-Display and Hot-Plug

`OnDisplaysChanged()` fires when a monitor is connected or removed. The `Info` struct contains EDID-parsed data: native resolution, supported modes, color primaries, HDR metadata. Scenic queries each display's native mode and creates a new Flatland layer stack per display. Layers and images are scoped to a `display_id`; a layer on display A cannot be committed to display B. Hot-plug removal atomically destroys all layers for the removed display.

G. Capture / Screenshot

The coordinator supports scanout capture via `ImportImageForCapture()` and `StartCapture()`. A VMO with `ImageType::CAPTURE` is registered; after `StartCapture()`, the display DMA engine reads the current scanout and writes it back to the capture VMO. This enables screenshots without GPU involvement and without disrupting the live scanout.

H. Color Management Pipeline in Detail

The degamma LUT has 1024 entries per channel (R, G, B), mapping encoded $[0,1] \rightarrow [0,1]$ linear. The CSC applies a 3×3 matrix plus pre- and post-offsets in linear light. This handles BT.709 $\rightarrow$ sRGB primaries conversion and wide-color-gamut mapping. The gamma LUT re-encodes to the output color space. On Intel hardware these correspond to the `PIPEDEGAMMA`, `PIPECSC`, and `PIPEGAMMA` register blocks. Fuchsia models them via `SetDisplayColorConversion()` and `SetDisplayGammaTable()` in the coordinator FIDL.

I. Coordinator State Machine

VI. Flatland: Modern 2D Compositor

A. Design Philosophy

Flatland replaces the deprecated 3D GFX API with a strictly 2D, hardware-friendly model:

- **Retained mode**: clients maintain scene state between `Present()` calls; only diffs are communicated.
- **Hardware-first**: Flatland maps scene graph nodes 1:1 to display hardware planes when possible.
- **Per-session isolation**: each client has its own Flatland instance with dedicated dispatcher thread.
- **Backpressure**: Present allowances prevent runaway clients from queuing unbounded frames.

B. Scene Graph Model

The scene graph is a **tree of Transforms**, each identified by a client-assigned `TransformId`. A transform carries a translation, scale, and optional clip rectangle. Content attachments are either an `Image` (pixel buffer from sysmem) or a `Viewport` (embedding another Flatland session's view). Only 2D affine transforms (translate + scale) are supported. This constraint enables direct mapping to display plane scaling hardware.

C. Session Linking and ViewTree

Flatland sessions are linked by exchanging `ViewCreationToken`/`ViewportCreationToken` handle pairs (Zircon eventpairs). The parent creates a `Viewport` referencing the token; the child creates a `View` from the matching token. Scenic stitches these into a global **ViewTree**. Input events are dispatched through this tree, with coordinate transformations tracked at each join.

The ViewTree is a forest (multiple roots are possible for multi-display setups). Each display has a scene root; the shell component's Flatland session is the first child. Application views are embedded as children of the shell's Viewport. The tree depth is typically:

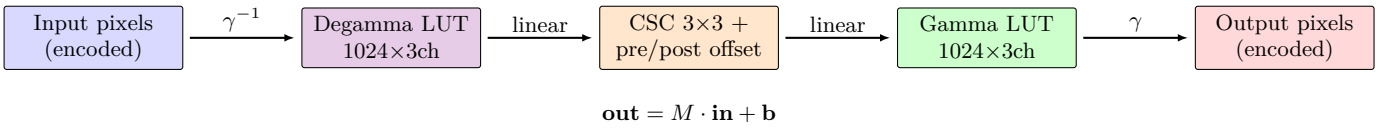
`scene-root` → `shell-view` → `app-viewport` → `app-view`

Each edge stores the cumulative affine transform (translation + scale) from the parent's coordinate space to the child's. When an input event arrives at screen coordinate (x, y), the Input Pipeline walks the ViewTree from root to leaf, applying the inverse of each transform to convert (x, y) into each View's local coordinate space until reaching the deepest View whose hit-test geometry contains the point.

The ViewTree also drives *focus*: the "focused" View receives keyboard events. Focus changes are propagated via `fuchsia.ui.focus/FocusChain` updates: Scenic publishes a new FocusChain (an ordered list of `ViewRef`s from root to focused leaf) whenever focus moves. The Input Pipeline uses this chain to route `KeyEvent`s to the correct `KeyboardListener`.

1) View Lifecycle and Visibility: A View is "attached" when its ViewCreationToken has been bound in a Flatland session and the matching Viewport exists in an ancestor session. A View is "detached" when its Viewport is removed or its parent session is destroyed. Detached Views are invisible (their content is pruned from the display-list) but retain their scene graph state. This mechanism is used by the shell to "park" suspended app views without destroying their scene graph. Resume is instant.

An attached View that has never called `Present()` does not appear on screen (it has no committed UberStruct). A View



$$\mathbf{out} = M \cdot \mathbf{in} + \mathbf{b}$$

Fig. 7. Display color management pipeline: out = Γ(CSC(Degamma(in))).

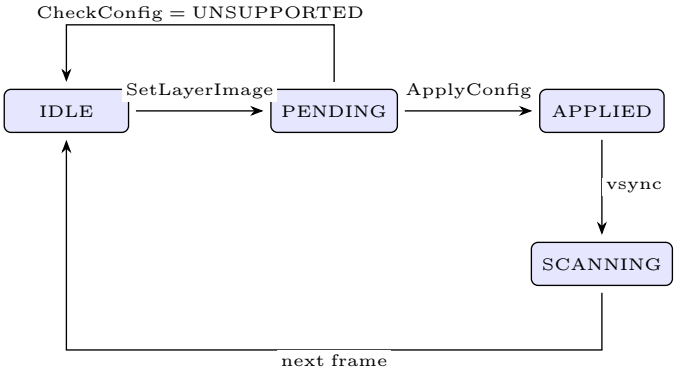


Fig. 8. Display coordinator config state machine.

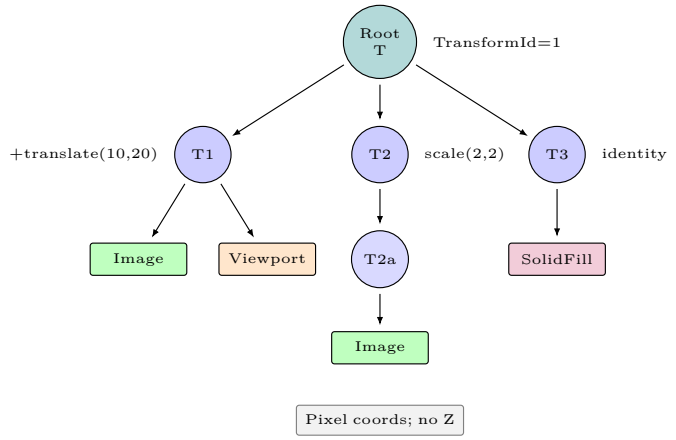


Fig. 9. Flatland scene graph: transform tree with image content and child viewport.

whose `Present()` acquire fences are perpetually unsignaled also does not appear on screen. Scenic renders the last successfully committed frame for that View, providing graceful degradation when an app hangs.

D. Image Import via Allocator

Listing 8. Allocator FIDL

```
closed protocol Allocator {
  strict RegisterBufferCollection(resource struct {
    export_token BufferCollectionExportToken;
    buffer_collection_token
        fuchsia.sysmem2.BufferCollectionToken;
  }) -> () error RegisterBufferCollectionError;
};
```

Flatland internally calls sysmem to set its own constraints (Vulkan image usage + display scanout requirements) before the collection is finalized. The app then calls `Flatland/CreateImage()` with a `BufferCollectionImportToken` to create a named image resource.

E. Present() Protocol and Backpressure

Listing 9. Flatland Present (abridged)

```
closed protocol Flatland {
  strict CreateTransform(struct { id TransformId; });
  strict SetTranslation(struct {
    id          TransformId;
    translation Vec2;
  });
  strict SetScale(struct {
    id    TransformId;
    scale VecF;
  });
  strict SetClipBoundary(struct {
    id     TransformId;
    region box2:optional;
  });
  strict SetContent(struct {
    id      TransformId;
    content ContentId;
  });
  strict SetImageBlendingFunction(struct {
    image_id  ContentId;
    blend_mode BlendMode;
    alpha      float32;
  });
  strict AddChild(struct {
    parent_transform_id TransformId;
    child_transform_id  TransformId;
  });
  strict Present(resource table {
    1: requested_presentation_time zx.Time;
    2: acquire_fences
         vector<zx.Handle:EVENT>:MAX;
    3: release_fences
         vector<zx.Handle:EVENT>:MAX;
    4: unsquashable bool;
  });
  -> OnNextFrameBegin(table {
    1: additional_present_credits uint32;
    2: future_presentation_infos
         vector<PresentationInfo>:MAX;
  });
  -> OnFramePresented(table {
    1: actual_presentation_time zx.Time;
    2: presentation_infos
         vector<PresentedFrameInfo>:MAX;
  });
};
```

Flow: (1) Client accumulates scene mutations. (2) Calls `Present()`, optionally specifying target time and fence handles. (3) Scenic signals `OnNextFrameBegin()` when the client may produce the *next* frame (backpressure credit). (4) After scanout, `OnFramePresented()` reports actual display time.

F. Hit Testing and Input Routing

Flatland maintains a parallel *hit-test geometry* tree alongside the render tree. When a touch or mouse event arrives at the Input Pipeline, it walks this tree to find the topmost View owning the hit point. Each Viewport has an associated `InputViewRef`, a Zircon eventpair handle identifying the owning component.

The coordinate transform chain is reconstructed from the transform tree: each transform's translation and scale is composed into a cumulative screen-to-view-local matrix. The touch point in screen coordinates is multiplied through the inverse chain to arrive at the target View's local pixel coordinates.

G. Clip Regions and Overdraw

Each Transform has an optional `clip_bounds` rect in the transform's parent coordinate space. This is used both for ren-

dering (clipping images at display-plane level, so the hardware plane only scans the clipped region) and for hit testing (touch events outside the clip do not reach children). Scenic's engine uses clip bounds to compute overdraw maps for occlusion culling: transforms entirely occluded by opaque content above them are pruned from the display-list.

H. Opacity and Blend Modes

`SetImageBlendingFunction(id, BlendMode, alpha)` configures Porter-Duff compositing:

- **SRC**: replace destination pixels (no blending; maps to hardware plane with pre-multiplied alpha disabled).
- **SRC_OVER**: composite over destination using per-pixel alpha (hardware plane alpha blending enabled).
- **SRC_ATOP**: composite only where destination has coverage. *This mode forces GPU compositing via Escher* since display planes do not implement `SRC_ATOP`.

A global opacity can also be set on a Transform; values below 1.0 force GPU compositing of the subtree.

I. UberStructSystem and Aggregation

Each Flatland session publishes its scene graph as an **UberStruct** at each `Present()` call. An UberStruct is a snapshot of the transform tree, content bindings, and acquire fence status. The `UberStructSystem` collects these snapshots atomically from all sessions. At frame deadline, the DisplayCompositor reads all UberStructs, merges them into a global display-list, and performs occlusion culling. This gives multi-session atomicity without locks between sessions: each session's snapshot is written once and read once per frame.

VII. Scenic GFX (Legacy 3D API)

The GFX API (deprecated, targeted for removal) provided a *3D retained-mode scene graph* supporting arbitrary node hierarchies with lighting, shadows, and cameras.

A. Resource Model

Resources were identified by client-assigned integer IDs within a Session:

- **Memory / Image / Buffer**: VMO-backed GPU objects.
- **Nodes**: `EntityNode` (transform group), `ShapeNode` (geometry + material leaf), `ViewNode` (session root), `CameraNode`.
- **Shape**: `Rectangle`, `RoundedRectangle`, `Circle`, `Mesh`.
- **Material**: `Texture` (Image) + color, opacity.
- **Scene**: root node + lights; required for rendering.
- **Camera**: `PerspectiveCamera` or `StereoCamera`.
- **Layer / LayerStack / Compositor**: rendering output chain.
- **View / ViewHolder**: cross-session embedding (3D-aware).

Operations were sent as commands inside `Session/Enqueue()` and committed via `Session/Present()`.

B. Differences from Flatland

Aspect	GFX	Flatland
Dimensionality	3D (full matrix)	2D (translate+scale)
Compositing	Always GPU (Escher)	HW-first, GPU fallback
Shadow/lighting	Yes (PBR, stencil)	No
Command model	Union vector	Individual methods
Status	Deprecated	Active / preferred

C. Lighting Model

The GFX API supported a PBR (physically-based rendering) lighting model through Escher: `AmbientLight` contributed a constant radiance term; `DirectionalLight` contributed a collimated source with a normalized direction vector and shadow-casting capability (stencil-shadow volumes in Escher); `PointLight` contributed a $1/r^2$ falloff source. Escher computed the lighting integral per ShapeNode material using metallic-roughness BRDF. This entire lighting pipeline is absent in Flatland.

D. Input Dispatch in GFX

GFX dispatched input events via `fuchsia.ui.input/InputEvent` on a separate channel per Session. Hit-testing was performed by casting a ray from the camera origin through the screen pixel and intersecting with ShapeNode bounding volumes in the 3D scene. The 3D intersection coordinate was then projected back to 2D for delivery to the hit View. Flatland replaces this with the 2D geometry tree hit-test described in §6.5.

E. GFX Session Protocol and Command Batching

The GFX API batched all operations into a `Session/Enqueue(vector<Command>)` call. The `Command` type was a FIDL union with 60+ variants:

Listing 10. GFX Session Enqueue (abridged)

```
type Command = flexible union {
  // Resource creation:
   1: create_resource CreateResourceCmd;
   2: release_resource ReleaseResourceCmd;
  // Node operations:
   3: add_child AddChildCmd;
   4: detach    DetachCmd;
   5: set_translation SetTranslationCmd;
   6: set_rotation    SetRotationCmd;  // Quaternion!
   7: set_scale       SetScaleCmd;
  // Material:
   8: set_material    SetMaterialCmd;
   9: set_texture     SetTextureCmd;
  // Camera:
  10: set_camera      SetCameraCmd;
  11: set_projection  SetProjectionCmd;
  // Lighting:
  12: add_light       AddLightCmd;
  13: set_light_color SetLightColorCmd;
  // ... 47 more variants
};
```

This union encoding required the Scenic server to dispatch on the variant tag for every command. That is $O(N)$ dispatch per frame, where N is the total command count. Flatland's individual FIDL methods allow the server to dispatch via the FIDL ordinal in the channel message header, eliminating the inner switch.

F. Migration from GFX to Flatland

Apps migrating from GFX to Flatland encounter three key differences:

1) **No 3D transforms**: GFX `SetRotation(quaternion)` and `SetScale(xyz)` become Flatland `SetScale(Vec2F)`. Rotation is not supported. Apps that used shallow 3D for card-flip animations must switch to GPU-rendered textures.
2) **No built-in lighting**: GFX's PBR lighting pipeline is absent in Flatland. Apps that used `DirectionalLight` for material highlighting must bake lighting into their textures or implement it in a Vulkan fragment shader.
3) **Present credits**: GFX's `Session/Present(uint64 present_time)` could be called without flow control.

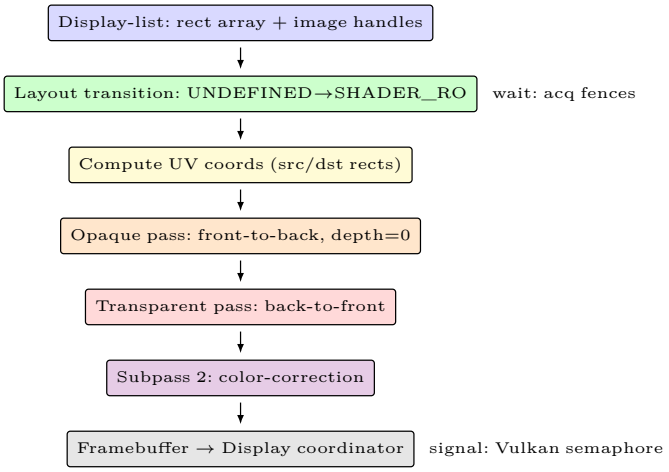


Fig. 10. Escher RectangleCompositor render pass stages.

Flatland enforces present credits; apps must implement the `OnNextFrameBegin` credit accounting to avoid session termination.

The Fuchsia SDK provides a `flatland_bridge` compatibility layer that translates a subset of GFX operations to Flatland, easing migration for simple 2D apps.

VIII. Escher: Vulkan Renderer

A. Role and Scope

Escher is a C++/Vulkan library at `src/ui/lib/escher/`. It is used by Scenic's **VkRenderer** when hardware compositing is insufficient. Escher provides: resource lifetime management, pipeline caching, render pass abstraction, staging buffer management, and GLSL→SPIR-V compilation via `shaderc`.

B. RectangleCompositor

The **RectangleCompositor** receives a flattened array of (source image, destination rect, blend mode) tuples from the Engine and produces a single composite image. Opaque rectangles are rendered front-to-back to maximize early-Z rejection; transparent rects are rendered back-to-front for Porter-Duff blending. A second Vulkan subpass applies color correction if the display hardware cannot do it natively. The subpass shares the framebuffer attachment, so pixels flow through the tile cache without a memory round-trip.

C. Synchronization with GPU

Escher maintains a per-frame `Frame` object with an associated Vulkan timeline semaphore. All GPU resources referenced in a frame hold a ref to the Frame object; they are only released after the semaphore is signaled, preventing VMO deallocation during GPU access.

D. Resource Management: Frame Object Lifecycle

`Escher::NewFrame()` allocates a `FramePtr` (ref-counted). All images, buffers, and pipelines used in the frame call `frame->TakeWhenSignaled(resource)`, which moves the resource ref into the Frame's deferred-release list. After `vkQueueSubmit` signals the timeline semaphore, an asynchronous waiter (a Zircon port wait on the semaphore event) fires. The waiter drops the `FramePtr`, releasing all held resources atomically. If a VMO's refcount hits zero at that point, the kernel reclaims physical pages immediately. This design avoids per-resource fence tracking: one semaphore per frame covers all resources.

E. Pipeline Caching and ShaderProgramFactory

`ShaderProgramFactory` maps `(shader_filename, preprocessor_defines)` → `ShaderProgram`. Each `ShaderProgram` lazily creates a `VkPipeline` on first use with a given render pass + vertex format combo. A Vulkan `VkPipelineCache` object (backed by a sysmem-pinned VMO) is serialized to disk on the first run and reloaded on subsequent runs, avoiding SPIR-V re-JIT on every boot.

F. Image Layout Transitions

Escher tracks the current `VkImageLayout` for all managed images in an `ImageLayoutTracker` map. `ImageLayoutTransition()` emits only the minimum necessary barriers: it computes the required pipeline stage masks and access masks from the (old layout, new layout) pair and inserts a `vkCmdPipelineBarrier` with those precise masks. This avoids the common mistake of using `VK_ACCESS_MEMORY_READ_BIT | VK_ACCESS_MEMORY_WRITE_BIT` for every barrier, which serializes the entire pipeline.

G. BatchGpuUploader

`BatchGpuUploader` batches all CPU→GPU staging uploads into a single command buffer submission. It manages a ring-buffer staging VMO mapped with write-combine cache policy (`ZX_CACHE_POLICY_WRITE_COMBINING`). CPU writes stream into the ring buffer without cache-flush overhead; the GPU reads via PCIe/NIC DMA once the buffer pointers are programmed. All staged resources are tracked in the Frame object and released after GPU completion.

H. Escher Shader Architecture

Escher organizes its shaders in a layered system:

- **Vertex shaders**: `flat_land_vertex.vert` is a minimal vertex shader that takes 2D rectangles (as `(x, y, w, h)` uniforms) and emits `gl_Position` and `texCoord` varyings. No matrix multiply; the 2D affine transform is inlined as two MAD instructions.
- **Fragment shaders**: `flat_land_rgb.frag` (RGB images), `flat_land_yuv.frag` (NV12/YUY2 with YCbCr→RGB conversion), `flat_land_solid_fill.frag` (solid color, no texture fetch). The YUV shader samples two Vulkan image planes (Y and UV) via `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM`.
- **Color correction shader**: `color_correction.frag` applies the 3×3 CSC matrix + gamma LUT as a post-process in subpass 2, reading the framebuffer as an input attachment (`layout(input_attachment_index=0)` in GLSL).

All Escher shaders are compiled from GLSL to SPIR-V at build time using `shaderc` and stored as C++ array literals (`spirv_cross`). At runtime, `vkCreateShaderModule()` ingests the SPIR-V; the ICD JITs it to GPU ISA on the first pipeline creation. Subsequent pipeline creations with the same SPIR-V reuse the GPU ISA from the `VkPipelineCache`.

I. Escher Memory Allocator: GpuAllocator

Escher's `GpuAllocator` wraps Vulkan memory allocation with sub-allocation:

1) For small allocations (<64KB): sub-allocate from a `GpuMemSlab`, a 4MB `VkDeviceMemory` block divided into fixed-size slots. This avoids the per-allocation overhead of `vkAllocateMemory()` (which is limited to 4096 calls on many IHV drivers).
2) For large allocations (≥64KB): allocate a dedicated `VkDeviceMemory` object. This is typical for framebuffers and swapchain images.
3) For sysmem-imported images: no allocation. The `VkDeviceMemory` is bound directly to the sysmem VMO via `VK_FUCHSIA_external_memory`.

The `GpuAllocator` tracks all allocations with their associated `Frame` objects, ensuring deferred release matches GPU completion.

J. RectangleCompositor: Vertex Layout

The vertex buffer layout for the RectangleCompositor is optimized for minimal vertex shader work:

Listing 11. Vertex layout for 2D rectangle rendering

```
struct FlatlandVertex {
  float position[2];  // NDC x, y in [-1, 1]
  float texcoord[2];  // UV in [0, 1]
  // Packed as: [x, y, u, v] = 4 x float32 = 16 bytes
};
// 4 vertices per rectangle (triangle strip):
// TL=(0,0,0,0), TR=(1,0,1,0), BL=(0,1,0,1), BR=(1,1,1,1)
// 6 rectangles share one VkDrawIndexed call via instancing;
// per-instance data: transform (4 floats), crop (4 floats),
//                    alpha (1 float), image index (1 uint).
```

Each rectangle is rendered as a single instanced draw call. The per-instance transform maps the unit-square vertex positions to the destination rect on screen; the per-instance crop maps UV coordinates to the source rect in the image. This packs all 2D compositing work into one draw call per image, minimizing Vulkan command overhead.

IX. Vulkan Integration

A. ICD Discovery

On Fuchsia, `libvulkan.so` is the standard Vulkan loader. It calls `fuchsia.gpu.magma/IcdLoaderDevice/GetIcdList()` to enumerate available vendor ICDs. The ICD is loaded into the application process; no kernel modules are involved.

B. Fuchsia-Specific Extensions

Listing 12. Key Fuchsia Vulkan extensions

```
VK_FUCHSIA_external_memory     // VMO <-> VkDeviceMemory
VK_FUCHSIA_external_semaphore  // zx::event <-> VkSemaphore
VK_FUCHSIA_buffer_collection   // sysmem collection -> Vulkan
VK_FUCHSIA_imagepipe_surface   // WSI: Flatland surface
```

`VK_FUCHSIA_buffer_collection` lets the Vulkan ICD participate in sysmem negotiation: the ICD sets `VkImageConstraintsInfoFUCHSIA` into the collection so allocated VMOs are guaranteed compatible with `VkImage` binding.

C. Imagepipe Surface and Swapchain

To render to screen, an app creates a `VkSurfaceKHR` via `vkCreateImagePipeSurfaceFUCHSIA()`, passing an imagepipe FIDL channel handle. The swapchain allocates a sysmem buffer collection with display + Vulkan constraints, creates `VkImage` objects bound to the VMOs, and cycles them through acquire/present with fence-mediated backpressure.

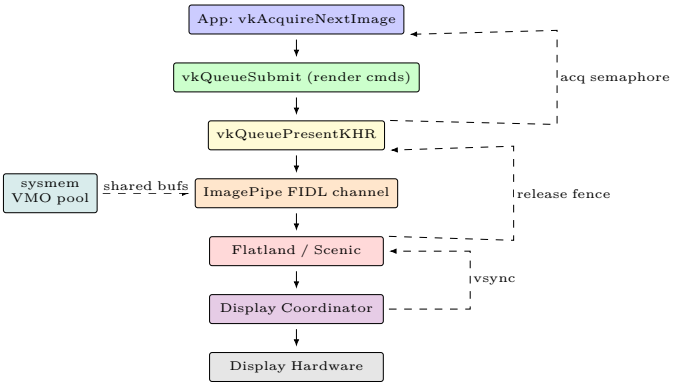


Fig. 11. Vulkan imagepipe swapchain flow with feedback signals.

D. VK_FUCHSIA_buffer_collection in Detail

Listing 13. Buffer collection Vulkan API flow

```
// 1. Import sysmem token into Vulkan
VkBufferCollectionCreateInfoFUCHSIA ci = {
  .sType = VK_STRUCTURE_TYPE_BUFFER_COLLECTION_CREATE_INFO_FUCHSIA,
  .collectionToken = sysmem_token_handle,
};
vkCreateBufferCollectionFUCHSIA(dev, &ci, NULL, &coll);

// 2. Set Vulkan image constraints into the collection
VkImageConstraintsInfoFUCHSIA ici = {
  .formatConstraintsCount = 2,
  .pFormatConstraints = fmt_constraints,  // e.g. BGRA32, NV12
  .bufferCollectionConstraints = buf_constraints,
  .flags = VK_IMAGE_CONSTRAINTS_INFO_CPU_READ_RARELY_FUCHSIA,
};
vkSetBufferCollectionImageConstraintsFUCHSIA(dev, coll, &ici);

// 3. sysmem allocates once all participants set constraints.
// 4. Create VkImage from negotiated collection slot
VkBufferCollectionImageCreateInfoFUCHSIA bci = {
  .sType = VK_STRUCTURE_TYPE_BUFFER_COLLECTION_IMAGE_CREATE_INFO_FUCHSIA,
  .collection = coll,
  .index = 0,  // buffer index within collection
};
image_info.pNext = &bci;
vkCreateImage(dev, &image_info, NULL, &image);

// 5. Bind device memory from collection
VkBufferCollectionPropertiesFUCHSIA props;
vkGetBufferCollectionPropertiesFUCHSIA(dev, coll, &props);
// props.memoryTypeBits indicates compatible memory types
vkAllocateMemory(...);  vkBindImageMemory(...);
```

E. Timeline Semaphores and Fuchsia Events

`VK_FUCHSIA_external_semaphore` allows exporting a Vulkan semaphore as a `zx::event` via `vkGetSemaphoreZirconHandleFUCHSIA()`. This event handle is then passed directly to `Flatland/Present()` as an acquire fence. The GPU signals the event at the end of the render pass (via the ICD programming a hardware semaphore signal instruction); Scenic waits on the event handle in its UberStructSystem before including the frame in aggregation.

F. Compute Pipelines and Async Compute

On Fuchsia, compute-only queues are supported via Magma's multi-context model. An async compute queue maps to a separate Magma context with its own GPU hardware queue (e.g., the AMD SDMA engine or Intel BCS engine). The ICD programs inter-queue semaphores using the standard Vulkan submission dependency mechanism; the MSD translates

these into hardware timeline semaphore waits across queue boundaries.

G. Vulkan WSI: Surface Types

Fuchsia supports two Vulkan WSI (Window System Integration) surface types:

- **ImagePipe surface** (`VK_FUCHSIA_imagepipe_surface`): connects to Flatland/Scenic. The app creates a Flatland session, allocates a buffer collection via the Allocator, creates a `VkSurfaceKHR` from the imagepipe channel, and presents via `vkQueuePresentKHR`. This is the standard path for windowed apps.
- **Display surface** (`VK_KHR_display`): takes exclusive control of the display coordinator. Used for fullscreen kiosk apps, VR headsets, and test harnesses. Not available while a compositor is running (the compositor holds the exclusive coordinator connection).

The swapchain is always triple-buffered internally: one buffer being scanned, one queued in Flatland, one being rendered. If the app calls `vkAcquireNextImage` with `timeout=0` and no buffer is available (all three are in-flight), it returns `VK_NOT_READY`. The correct response is to wait for `OnFramePresented` before acquiring again.

H. Vulkan Memory Types on Fuchsia

Fuchsia exposes two primary Vulkan memory type categories:

- **Device-local, host-invisible**: maps to VMOs in a device-local heap (e.g., `GOLDFISH_DEVICE_LOCAL` in emulator, or GPU-local SRAM on custom SoCs). Fastest for GPU reads/writes; CPU cannot access.
- **Host-visible, device-coherent**: maps to VMOs in `SYSTEM_RAM` heap with `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`. On Fuchsia, host-coherent memory uses `CACHED` VMO policy on cache-coherent platforms (ARM with CCI-500) and requires explicit `vkFlushMappedMemoryRanges()` on non-coherent platforms. Coherency is reported per-platform in `VkPhysicalDeviceMemoryProperties`.

Fuchsia does not have a dedicated "HOST_CACHED" memory type separate from "HOST_COHERENT". The distinction is expressed instead via the sysmem `cache_policy` constraint. The ICD maps the VMO with the policy specified by sysmem after `WaitForAllBuffersAllocated()`.

I. SPIR-V Compilation and Caching on Fuchsia

On Fuchsia, SPIR-V→ISA compilation happens inside the ICD (`libvulkan_vendor.so`). The compiled ISA blobs are cached using Vulkan's `VkPipelineCache` mechanism. The cache is stored in a VMO and can be serialized to persistent storage via `vkGetPipelineCacheData()`. On subsequent runs, the cached ISA is reloaded with `vkCreatePipelineCache(initialDataSize=..., pInitialData=...)`. This is critical for Fuchsia boot performance: without a populated pipeline cache, shader compilation at app startup can stall for hundreds of milliseconds on a complex 3D app.

Fuchsia's component framework provides a `fuchsia.io/Directory` capability that apps use to persist the pipeline cache binary between sessions. The cache is keyed on the ICD's vendor ID + device ID + driver version, so it is invalidated automatically on driver updates.

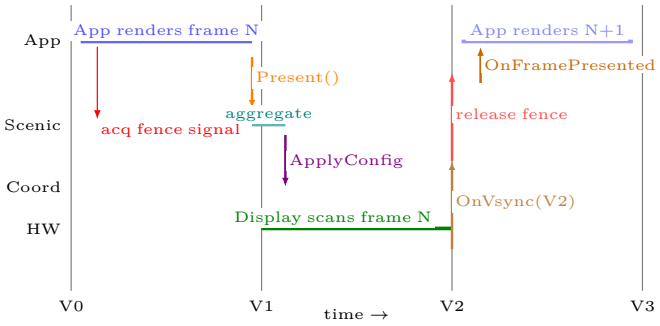


Fig. 12. Frame scheduling timeline: vsync, Present(), fences.

X. Frame Scheduling

A. End-to-End Timeline

B. Vsync-Driven Model

The **FrameScheduler** wakes up slightly before each vsync deadline, budgeting: UberStruct aggregation (∼0.5ms), engine flatten + occlusion culling (∼0.5ms), `ApplyConfig()` round-trip (∼0.5ms). Total budget: one vsync period minus 2ms guard time.

C. Acquire and Release Fences

- **Acquire fences** (client → Scenic): Zircon events the client signals after GPU rendering. Scenic's UberStructSystem waits for all acquire fences before including the frame. If not signaled by deadline, the frame is deferred to the next vsync.
- **Release fences** (Scenic → client): signaled after the display controller confirms the old image is no longer scanned (vsync N+1 after new config becomes active). Client reuses the buffer only after this signal.

D. PresentationInfo and Latency Control

`OnNextFrameBegin()` delivers `FuturePresentationInfos`: predicted vsync timestamps and latch deadlines for upcoming frames. Latency-sensitive apps target the *nearest* presentation time; video players pipeline 2–3 frames. The anti-drift formula:

$$t_{\text{req}} \leftarrow \max(t_{\text{req}},\ t_{\text{prev\_req}}), \quad t_{\text{req}} = t_{\text{vsync}} - \tfrac{1}{2}T_{\text{vsync}}$$

E. FrameScheduler Algorithm

The FrameScheduler implements a vsync-locked admission algorithm:

1) On `OnVsync(t)`: update predicted next vsync $\hat{t}_{n+1}$ via IIR filter: $\hat{t}_{n+1} = \alpha \cdot (t - t_{n-1}) + (1-\alpha) \cdot \hat{T}$, where $\alpha = 0.1$ and $\hat{T}$ is the nominal period.
2) Walk the pending Present queue: for each Present with $t_{\text{req}} \leq \hat{t}_{n+1}$, move it to the "ready" set.
3) If any ready Present's acquire fences are all signaled: schedule aggregation.
4) If $\hat{t}_{n+1} - t_{\text{now}} < t_{\text{guard}}$ (2ms): defer remaining Presents to the next vsync.
5) After aggregation, if the hardware config changed: call `ApplyConfig(ConfigStamp++)` to commit it.
6) `ConfigStamp` is echoed back in `OnVsync` confirming which config is active on screen.

F. Squashing

If a client calls `Present()` twice before the compositor processes the first, and neither is marked `unsquashable`, the compositor *squashes* them: only the later scene state is applied; both frames' acquire fences are waited on; all release fences are signaled together. This avoids wasted GPU render work for frames that will never appear on screen.

G. Jank Detection

If a Present's acquire fences are not signaled by the latch deadline (vsync − guard), Scenic logs a jank event. `OnFramePresented`'s `presentation_infos` includes `actual_presentation_time` vs. `requested_presentation_time`; a delta > 0.5 frame periods triggers a jank trace event visible in `fx trace` output.

H. Display Pipeline Latency Analysis

The total pipeline latency (app render start → photon emission) has two components:

Fixed latency (irreducible): the display controller's scanline DMA requires all pixels to be written to the VMO before the first scanline is read. At 1080p 60Hz, scanning takes 16.7ms. The first pixel is emitted $T_{\text{blank}} = 1$–2ms after the vsync (during which the display scans the first active line from memory). Total fixed latency: $\approx T_{\text{frame}} + T_{\text{blank}} \approx 18$ms.

Variable latency (from pipeline depth): if the app renders N frames ahead of vsync, an additional $N \times T_{\text{frame}}$ latency is incurred. With triple-buffering ($N = 1$, one queued frame), the total is ≈ 35ms at 60Hz.

The FrameScheduler allows apps to reduce N to 0 (no pre-rendered frames in the queue) by targeting the *current* vsync as the requested presentation time. This minimizes latency to ≈ 18ms at 60Hz at the cost of higher jank probability (the GPU render must complete within the current frame period).

$$L_{\text{total}} = T_{\text{render}} + N \cdot T_{\text{frame}} + T_{\text{blank}}$$

For stylus-critical applications (digital ink), Fuchsia recommends $N = 0$ plus stylus prediction to achieve effective latency below the human perception threshold of ≈ 10ms.

I. Multi-Vsync Pipelining for Video

Video decoders produce frames at 24/30/60fps. When the display runs at 60Hz and the video is 24fps, Scenic must hold each video frame for 2–3 vsync periods. The Flatland `requested_presentation_time` field enables precise frame-to-vsync alignment: the video player computes the PTS (Presentation Timestamp) for each frame, converts it to Zircon monotonic time, and sets `requested_presentation_time` to the target vsync. The FrameScheduler queues the frame and presents it exactly at that vsync, achieving correct 3:2 pulldown cadence without tearing.

XI. Memory Model

A. VMO as Universal Currency

Every GPU/display buffer is a Zircon VMO. VMO permissions: **CPU mapping** via `zx_vmar_map()`; **GPU mapping** via Magma `ImportBuffer()`; **DMA pin (PMT)** via `zx_bti_pin()` giving a physical address range stable until `zx_pmt_unpin()`.

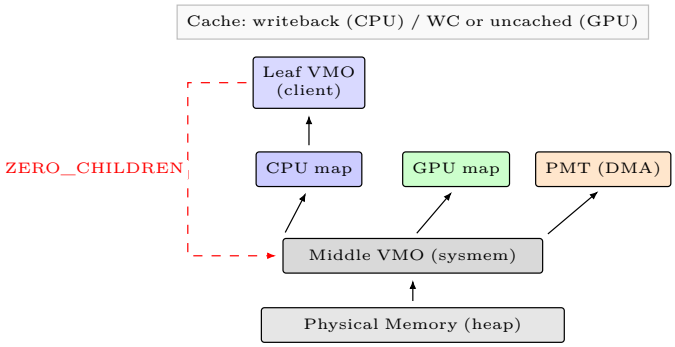


Fig. 13. VMO hierarchy: heap → middle → leaf; PMT for DMA pinning.

B. Cache Coherence

- CPU writes (command buffer upload) must be cache-flushed before GPU DMA reads them on non-coherent AXI ports.
- On ARM SoCs with non-coherent GPU: ICD calls `zx_cache_flush()` before `ExecuteCommandBuffers()`; GPU signals IRQ on read completion.
- Display DMA on most SoCs is cache-coherent (reads from LLC); no flush needed between GPU render and display scanout.
- Secure/protected memory (`AMLOGIC_SECURE` heap) is *CPU-inaccessible*: only the video decoder and display DMA port can access it.

C. Heap Types Summary

Heap	Properties	Use case
`SYSTEM_RAM`	CPU+GPU, cached	General textures
`CONTIGUOUS`	Phys. contiguous	HW w/o IOMMU
`AMLOGIC_SECURE`	CPU-inaccessible	DRM video
`GOLDFISH_DEVICE_LOCAL`	VRAM on emulator	Emulator only
`tee_secure`	TEE-controlled	Enc. content

D. IOMMU Integration

On platforms with an IOMMU, the MSD obtains a Bus Transaction Initiator (BTI) handle from the platform device node via `fuchsia.hardware.platform.device/PlatformDevice.GetBtiById()`. When the MSD calls `zx_bti_pin()` on a GPU command buffer VMO, the kernel programs the IOMMU to allow DMA only from those specific physical pages to the GPU's bus address space. A compromised GPU cannot DMA to arbitrary physical memory. The IOMMU sandboxes its address space just as the MMU sandboxes a CPU process.

E. Protected Memory and DRM

The full DRM video flow: (1) Video decoder opens a sysmem collection with `AMLOGIC_SECURE` heap. (2) TEE allocates physical pages inside the carve-out; TZC-400 programs bus-level access control. (3) Display coordinator DMAs from these pages via its secured DMA port. (4) GPU can read them only inside a *protected context* (Mali protected mode), requested by the MSD when the Vulkan app uses `VkDeviceMemory` from a protected sysmem collection.

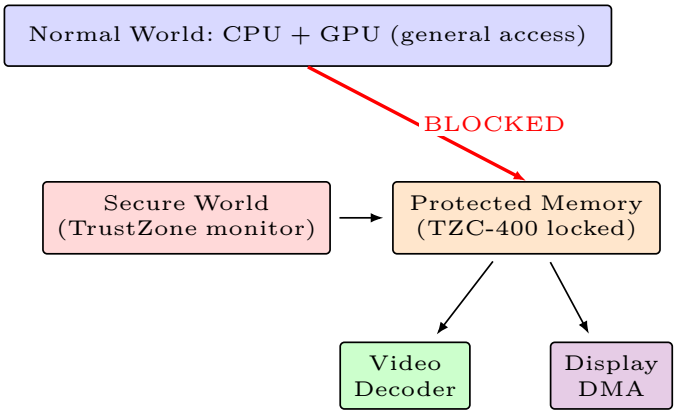


Fig. 14. TrustZone memory protection: CPU/GPU cannot access protected memory; only video decoder and display DMA can.

F. Memory Pressure and Eviction

Zircon signals memory pressure via `ZX_INFO_KMEM_STATS_EXTENDED` and a `fuchsia.memorypressure/Watcher` protocol. sysmem can receive `OnLevelChanged()` events and zero pages for purgeable VMOs. The ICD should mark pipeline caches and descriptor pool backing VMOs as purgeable (`ZX_VMO_OP_COMMIT` / `ZX_VMO_OP_DECOMMIT`); when purged by the kernel, the ICD recreates these caches on demand.

G. GPU Memory Tiling and Compression

Modern GPUs use tiled memory layouts to improve cache locality during rendering:

- **Linear**: rows of pixels stored sequentially. Required by most display controllers for scanout. Address: byte $= y \times \text{pitch} + x \times \text{bpp}$.
- **Tile-X (Intel)**: 512×8 pixel tiles (for 32bpp). Address: byte = (tile_row × surface_width_in_tiles + tile_col) × 4096 + inner_offset. Reduces row-stride miss rate in texture samplers.
- **Tile-Y (Intel)**: 128×32 pixel tiles. Higher cache efficiency for 3D rendering; not compatible with display scanout on older hardware.
- **Mali AFBC (Arm Framebuffer Compression)**: losslessly compresses tiles using superblock headers. Reduces memory bandwidth for the display DMA read by 30–50% on typical UI content. sysmem exposes AFBC via the `format_modifier = ARM_AFBC_16x16` DRM modifier.

The sysmem constraint `format_modifier` field carries the DRM modifier for tiled/compressed formats. When all participants agree on AFBC, sysmem allocates the VMOs with AFBC superblock headers; the GPU renders into AFBC, and the display DMA reads AFBC natively. Both sides avoid a linearization pass.

H. Large Buffer Allocation: Contiguous Pools

For buffers requiring physical contiguity (display planes on SoCs without IOMMU, or video decoders with limited address bit width), sysmem uses the `SysmemContiguousPool`:

1) At boot, the bootloader reserves a physically contiguous region (e.g., 128MB) and passes its base address and size to Fuchsia via the ZBI (Zircon Boot Image).
2) sysmem reads the ZBI item `ZBI_TYPE_MEM_CONFIG` and creates the contiguous pool from the reserved range.
3) When a buffer collection requests `is_physically_contiguous = true`, sysmem's contiguous pool allocator carves out a sub-range. The physical base address of the allocated VMO is accessible to the display driver via `zx_bti_pin()`.
4) Fragmentation is managed with a buddy allocator; after display shutdown, the pool is released back to the general heap.

I. Shared Memory Between CPU and GPU: Cache Policy

The cache policy of a VMO's mapping critically affects correctness and performance:

Policy	CPU writes	GPU reads
`CACHED`	Write-back (fast)	Requires `zx_cache_flush` on non-coherent GPU
`WRITE_COMBINING`	Write-combine (streaming)	No cache; GPU reads directly from DRAM
`UNCACHED`	Uncached (slow)	No cache; used for MMIO-like regions

Command buffers are typically mapped `WRITE_COMBINING` on the CPU side: the CPU streams commands into the ring buffer without polluting the L1/L2 cache, and the GPU DMA engine reads from DRAM without an IOMMU cache-coherency protocol. Image buffers intended for CPU-side software rendering use `CACHED` with an explicit `zx_cache_flush(ZX_CACHE_FLUSH_DATA | ZX_CACHE_FLUSH_INVALIDATE)` before the GPU reads them.

J. Buffer Lifecycle and Deallocation

sysmem tracks all references (VMO handles, mappings, PMTs). A buffer returns to the heap only when all three are clear: (1) Client closes BufferCollection channel and VMO handles. (2) Scenic destroys its ImageId. (3) GPU removes mapping (`ReleaseBuffer()`). (4) Display coordinator calls `zx_pmt_unpin()` after removing its layer. (5) Kernel fires `ZX_VMO_ZERO_CHILDREN` on middle VMO. (6) sysmem reclaims to heap.

XII. Fuchsia Driver Framework and MSD Hosting

A. DFv2 Overview

The Fuchsia Driver Framework v2 (DFv2) loads drivers as ELF shared libraries (`.so`) into a `driver_host` process. The `driver_host` is a privileged process whose capabilities are granted by the Driver Manager at bind time:

- **MMIO access**: a handle to the GPU's PCI BAR or platform MMIO range, mapped via `MapMmio()`.
- **Interrupt handle**: a `zx::interrupt` bound to the GPU's IRQ, waited by the MSD's IRQ thread.
- **BTI handle**: a bus transaction initiator for DMA pinning (IOMMU integration).
- **Outgoing directory**: the `driver_host` exposes the `fuchsia.gpu.magma/Device` protocol here for discovery by the Vulkan loader.

B. Bind Rules

A `.bind` file expresses which device the MSD attaches to:

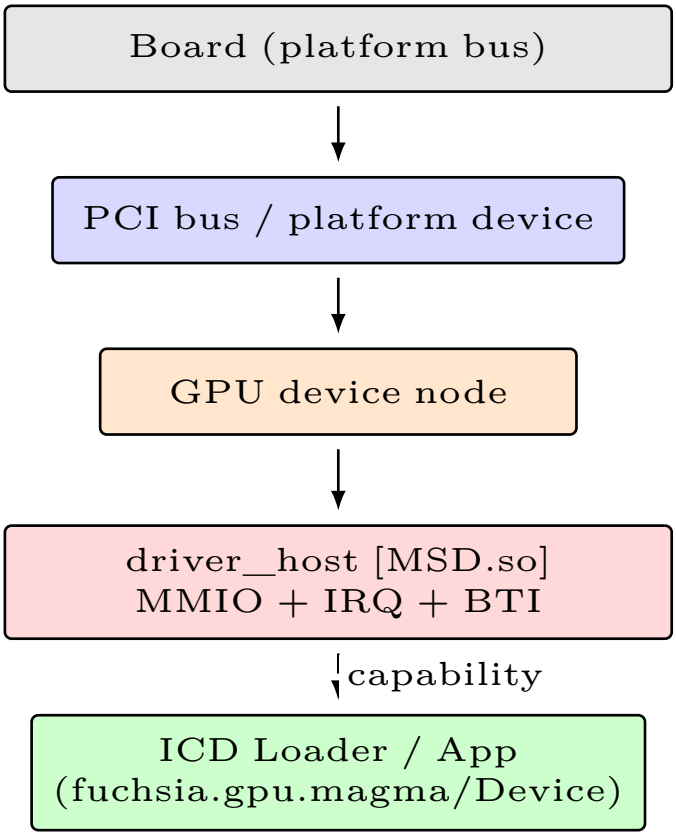


Fig. 15. DFv2 device topology: GPU driver_host exposes Magma Device protocol.

Listing 14. Intel GPU bind rule (simplified)

```
using fuchsia.pci;
fuchsia.pci.BIND_PCI_VID == 0x8086;
fuchsia.pci.BIND_PCI_CLASS == 0x03;  // Display controller
accept fuchsia.pci.BIND_PCI_DID {
  0x3185,  // Intel UHD 610
  0x3e92,  // Intel UHD 630
};
```

The Driver Manager matches bind rules against device properties in the device topology tree and selects the appropriate MSD at driver enumeration time.

C. Outgoing Directory and Service Discovery

The `driver_host` serves its protocols in a VFS "outgoing directory" exposed via `fuchsia.io/Directory`. The Vulkan loader connects to `fuchsia.gpu.magma/Device` by opening `/dev/class/gpu/000`. This path is a symlink managed by devfs into the driver host's outgoing directory. No global ambient authority is needed: the app must hold a channel to devfs to enumerate GPUs, and devfs enforces access control by component manifest.

D. Driver Host Sandboxing

The `driver_host` process is sandboxed by Zircon's job policy mechanism:

- The driver host's job has `ZX_POL_NEW_PROCESS = DENY`, preventing the driver from forking child processes.
- The driver host's job has `ZX_POL_AMBIENT_MARK_VMO_EXEC = DENY`, preventing JIT compilation within the driver.
- The MMIO region VMO is mapped with `ZX_VM_PERM_READ | ZX_VM_PERM_WRITE` but *not* `ZX_VM_PERM_EXECUTE`, preventing the driver from accidentally executing GPU register data.
- The driver host cannot open arbitrary file paths. It receives only the capabilities explicitly granted by the Driver Manager (MMIO handle, interrupt handle, BTI handle, and its own outgoing directory).

This sandboxing means a bug in the MSD (e.g., a stack overflow from a malformed command buffer) can crash the `driver_host` process without affecting other components. The Driver Manager detects the crash and optionally restarts the driver host, reconnecting all MSD clients via the epitaph/reconnection protocol.

E. DFv2 Driver Lifecycle

1) **Driver enumeration**: At boot, the Driver Manager reads the driver index (a list of `.bind` files and their corresponding `.so` paths from the boot filesystem). It matches each device node in the topology against the bind rules.
2) **Driver host creation**: When the GPU device node is matched, the Driver Manager creates a new `driver_host` process, grants it the GPU's MMIO/IRQ/BTI capabilities, and loads the MSD `.so` into it via `dlopen()`.
3) **Driver start**: The MSD's `Start()` entry point is called with a `StartArgs` struct containing the capability handles. The MSD initializes the GPU, maps MMIO, starts the IRQ thread, and calls `outgoing->AddProtocol<Device>(this)` to expose the Magma Device protocol.
4) **Client connection**: The Vulkan loader (in the app process) opens `/dev/class/gpu/000` via devfs and calls `GetIcdList()`, then `Connect2()`. The MSD creates a per-client connection object with its own Primary channel handler thread.
5) **Driver stop**: If the GPU is hot-removed or the driver is being updated, the Driver Manager calls `PrepareStop()`. The MSD sends epitaphs to all active Primary channels, waits for client connections to close, unmaps MMIO, and calls `Stop()`.

F. GPU Firmware and Microcode Loading

Most GPU vendors require firmware (microcode) to be loaded into the GPU before it can execute command buffers. On Fuchsia, firmware loading is handled by the MSD using the `fuchsia.boot.items/BootItems` protocol or the `fuchsia.driver.compat/DeviceServer` VNODE mechanism:

- **Mali GPU**: The firmware binary (`mali_csffw.bin`) is read from the boot filesystem, placed into a physically contiguous VMO, and DMA'd to the GPU's CS firmware region via the `GPU_CONTROL.CSF_FW_ADDRESS` register.
- **Intel GPU**: The GuC (Graphics Microcontroller) firmware is loaded by writing its physical address to `GUC_GGTT_OFFSET` and triggering a firmware load via `GUC_WOPCM_SIZE`. The GuC handles GPU context save/restore for hardware preemption.
- The firmware VMO is pinned via the BTI (PMT) to ensure the physical pages remain stable during the GPU DMA load sequence.

XIII. Input Pipeline and Display Interaction

A. Input Stack Overview

Input in Fuchsia flows from HID hardware through a chain of normalizing components to the focused View. The pipeline:

1) **HID driver**: reads from USB/I2C HID device; publishes `fuchsia.input.report/InputReport` structs.

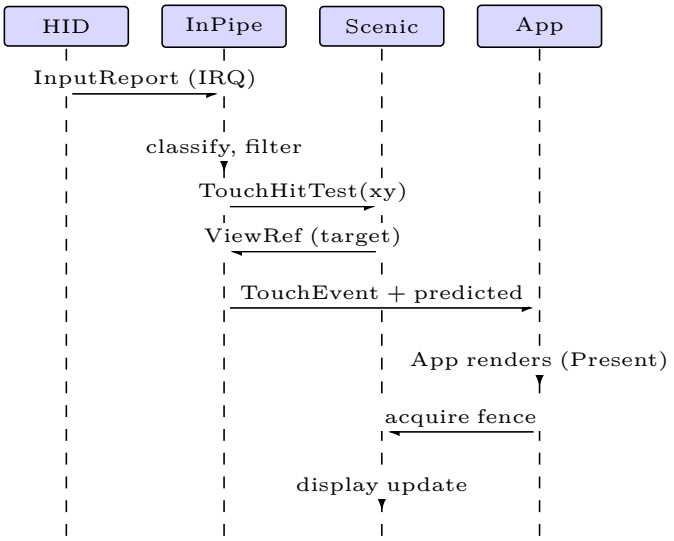


Fig. 16. Input pipeline: HID event to display update sequence.

2) **Input Pipeline component**: consumes `InputReport`, applies dead-zone filtering, pointer classification (touch vs. stylus vs. mouse), and gesture detection.
3) **ViewTree hit-test**: the Input Pipeline calls Scenic's `fuchsia.ui.input3/TouchHitTest` to find the target View for the event's screen coordinate.
4) **TouchSource delivery**: the event is sent to the target View's `fuchsia.ui.pointer/TouchSource` channel.
5) **App response**: app processes the event, possibly calls `Flatland/Present()` with updated scene.

B. Pointer Classification and Gesture Arena

The Input Pipeline runs a *gesture arena* for multi-touch events. When multiple Viewports overlap at a touch point, each candidate View is entered into the arena. Views declare their interest in a touch event sequence by calling `TouchSource/Watch()` with a `TouchResponse` of `MAYBE` (interested but not yet claiming). The arena resolves by:

1) **Prioritization**: Deeper Views (closer to the leaf in the ViewTree) have higher priority.
2) **Claim**: When a View responds `YES` to a touch event, it claims the gesture. All other Views receive a final event with `interaction_view_hit` set to false, allowing them to clean up.
3) **Timeout**: If no View claims within 50ms, the event is dispatched to the highest-priority View.

This arena model enables swipe-from-edge gestures to be intercepted by the shell (which sits above app Views in priority) without requiring apps to release their touch tracking explicitly.

C. High-Refresh-Rate (120Hz) Input

At 120Hz, the vsync period is 8.33ms. The Input Pipeline polls HID devices at 1ms intervals (higher than the 8.33ms frame rate) to minimize touch sampling latency. Sampled touch points between frames are delivered as `actual_samples` in a single `TouchEvent`. Apps can choose to render at 120fps (consuming all samples) or 60fps (consuming every other vsync), or implement a sub-frame curve renderer that interpolates between `predicted_samples`.

For stylus input, the HID report rate is often 240Hz (4ms period). The Input Pipeline delivers all stylus reports since the last frame as a vector of `actual_samples`; the app draws a Bézier curve through all of them in one GPU render pass, achieving sub-pixel stylus fidelity.

D. Latency Budget

At 60Hz, the end-to-end latency budget for touch-to-pixel is 16.7ms. Typical breakdown:

- HID interrupt → `InputReport`: ∼1ms (driver latency).
- Input Pipeline normalize + hit-test: <0.5ms.
- App event handler + `Present()`: ∼2ms.
- Scenic aggregation + `ApplyConfig()`: ∼1.5ms.
- Remaining budget for GPU render: ∼11ms.

At 120Hz (8.33ms budget per frame), the GPU render budget shrinks to ∼4ms. This requires apps to use GPU compute shaders rather than fragment shaders for UI rendering (lower setup overhead per draw call), and to minimize Vulkan render pass count (one render pass per frame for the entire UI).

E. Stylus Prediction

The `fuchsia.ui.pointer/TouchSource` protocol delivers events as `TouchInteractionResult` tables. Each result includes:

- **actual_samples**: measured touch points for the current frame.
- **predicted_samples**: ML-predicted future touch points for the next 1–2 frames.

Apps can pre-render the predicted stroke position and display it immediately, then correct on the next `Present()` when actual samples arrive. This technique reduces perceptual stylus latency to near zero even at 60Hz.

F. Accessibility and the Magnification Pipeline

Fuchsia's accessibility framework intercepts the display output to apply screen magnification and color filters. The accessibility manager holds a Flatland session that inserts itself between the shell and the display coordinator:

1) The accessibility manager creates a root Transform that applies a scale + translation matrix, implementing magnification.
2) Color-blind assistance filters (deuteranopia, protanopia) are applied via `SetDisplayColorConversion()` on the display coordinator. A 3×3 Daltonization matrix is set as the CSC, which requires no GPU rendering.
3) Screen-reader semantics (focus rings, content labels) are delivered via a separate `fuchsia.accessibility.semantics/SemanticsManager` channel, parallel to the Flatland visual tree.

The magnification transform is applied at the Flatland layer (not in the display hardware), so the magnified content is GPU-composited by Escher's RectangleCompositor. This means magnification forces GPU compositing mode regardless of the hardware plane availability.

G. Key Input Protocols

Listing 15. Input protocols (abridged)

```
// Keyboard events:
closed protocol KeyboardListener {
  strict OnKeyEvent(struct {
    event    KeyEvent;
```

```
  }) -> (struct { handled Handled; });
};
// Touch and pointer:
closed protocol TouchSource {
  strict Watch(struct {
    responses vector<TouchResponse>:MAX;
  }) -> (struct {
    events vector<TouchEvent>:MAX;
  });
};
struct TouchEvent {
  timestamp       zx.Time;
  actual_samples  vector<PointerSample>:MAX;
  predicted_samples vector<PointerSample>:MAX;
  view_parameters ViewParameters;  // coord transform
};
```

XIV. Coordinate Systems, DPI, and Logical Pixels

A. Physical vs. Logical Pixels

Fuchsia distinguishes *physical pixels* (hardware pixels on the display panel) from *logical pixels* (coordinate units used by app layout engines). The conversion factor is the *device pixel ratio* (DPR), which equals the display's DPI divided by a platform-defined reference DPI (typically 96 DPI for desktop, 160 DPI for mobile):

$$\text{DPR} = \frac{\text{display DPI}}{\text{reference DPI}}$$

A 4K (3840×2160) display at 288 DPI has DPR=3. At DPR=3, an app operating in logical pixels uses a 1280×720 logical canvas; each logical pixel maps to a 3×3 block of physical pixels.

B. Coordinate Spaces in the Stack

The Fuchsia graphics stack uses multiple coordinate spaces:

- **Screen space**: physical pixels, origin top-left. Used by the display coordinator (layer positions, vsync timing).
- **Scene space**: Flatland's coordinate system. When a View is attached at the root, its scene space equals screen space. After transforms (translate, scale), scene space differs from screen space.
- **View-local space**: each Flatland View has its own local coordinate origin. The `ViewParameters` struct in TouchEvent gives the transform from view-local to scene space.
- **Logical space**: app-level coordinate space, typically DPR-adjusted. A Flutter app at DPR=2 renders to a `VkImage` at 2× logical resolution; the Flatland transform scales it back down by 1/DPR to fill the screen.
- **Texture space**: UV coordinates in $[0, 1]$ over the source image. Used by Escher's vertex shader to sample the correct sub-rect of the image.
- **NDC** (Normalized Device Coordinates): Vulkan's clip-space $[-1, 1]^2$. The RectangleCompositor maps destination rects in screen space to NDC for the vertex shader.

C. DPI Notification and Dynamic Scaling

When the display DPI changes (external monitor connected, display mode switched), Scenic fires a `ViewRef.OnMetrics()` event carrying the new `ViewportMetrics` (logical size, DPR). The app receives this event, reconstructs its swapchain at the new resolution, and resubmits a `Present()` with the updated image size. The entire resize is buffer-copy-free: the sysmem collection is re-negotiated with the new dimensions, the old VMOs are released, and new VMOs are allocated.

D. Sub-Pixel Rendering and Fractional Scaling

When DPR is non-integer (e.g., 1.5 for a 144 DPI display), apps face fractional scaling. Fuchsia's recommendation: round logical coordinates to the nearest *half-pixel* (0.5 physical pixels) to avoid sub-pixel rendering artifacts. Flutter's engine rounds text baseline positions to the nearest physical pixel to ensure crisp text at all DPR values.

For display coordinator hardware that supports fractional plane scaling (`SetLayerPrimaryPosition()` with non-integer `src_frame` / `dest_frame` ratios), Flatland can request hardware bilinear scaling of the source image to the destination rect. This is used for the 1.5× DPR case: the app renders at 2× DPR, and the display hardware scales 2/3 in both dimensions to reach the physical display resolution.

XV. HDR, Wide Color Gamut, and Color Science

A. Color Spaces in Fuchsia

Fuchsia's graphics stack handles color management end-to-end from pixel buffer to display output. The color pipeline must handle:

- **sRGB**: standard 8-bit per channel color space used by UI content. Gamma encoding: $\Gamma \approx 2.2$ (exact piecewise formula per IEC 61966-2-1). Primary chromaticities: D65 white point, BT.709 primaries.
- **Display P3**: wider gamut (25% more than sRGB), used by modern OLED and LCD panels. Same D65 white point, P3 primaries.
- **BT.2020**: very wide gamut, used in HDR video. Primary chromaticities reach deeper into the visible spectrum than any commercial phosphor/LED; content maps into this space via PQ or HLG transfer functions.
- **scRGB (linear)**: linear float16 color space; HDR headroom up to 80 nits per unit (so values > 1.0 represent super-white HDR content). Used as the internal compositing color space in Scenic's HDR mode.

sysmem's `ColorSpace` enum expresses the color space of pixels in a buffer: `SRGB`, `REC709`, `REC2020`, `PASSTHROUGH`. The display coordinator reads this from the `ImageConfig` to configure the degamma LUT and CSC matrix for the correct linearization + gamut mapping pipeline.

B. HDR Metadata and EDID Parsing

The display coordinator's `Info` struct (returned in `OnDisplaysChanged`) includes HDR metadata parsed from the EDID/CEA-861 extension:

Listing 16. Display HDR info FIDL (abridged)

```
type HdrMetadata = struct {
  // Maximum display luminance (nits):
  max_luminance float32;
  // Maximum frame-average luminance:
  max_full_frame_luminance float32;
  // Minimum display luminance:
  min_luminance float32;
  // Supported electro-optical transfer functions:
  eotf                    bits {
    TRADITIONAL_SDR = 0x01;
    TRADITIONAL_HDR = 0x02;
    PQ              = 0x04;  // SMPTE ST.2084
    HLG             = 0x08;  // BT.2100 HLG
  };
  // Display color primaries (CIE xy chromaticities):
  primaries ColorPrimaries;
};
```

When the display reports PQ HDR support, Scenic can activate HDR mode: the compositing color space switches to scRGB (linear float16), the degamma LUT is programmed with

the PQ inverse EOTF (ST.2084 curve), and the gamma LUT is set to identity (the display applies its own PQ decode). Content that declares `ColorSpace::REC2020` in its sysmem constraints bypasses the Escher color correction subpass and goes directly to the display's PQ pipeline.

C. ICC Profiles and Wide-Color Gamut Mapping

For displays advertising Display P3 or BT.2020 primary support, Scenic computes a gamut mapping CSC matrix at initialization:

$$M_{\text{gamut}} = M_{\text{XYZ}\to\text{P3}} \cdot M_{\text{BT709}\to\text{XYZ}}$$

This 3×3 matrix converts from the input color space (typically sRGB/BT.709) to the display's native primaries via the CIE XYZ intermediate space. The matrix is programmed into the display coordinator's `SetDisplayColorConversion()` call. UI content (8-bit sRGB) is automatically gamut-mapped to the wider display gamut without any GPU work.

For content that provides its own ICC profile (e.g., an image with an embedded AdobeRGB profile), Scenic computes the additional matrix $M_{\text{AdobeRGB}\to\text{P3}}$ and applies it in Escher's color correction subpass before the display-coordinator-level CSC.

D. Tone Mapping for HDR Content on SDR Displays

When HDR content (PQ-encoded, peak luminance > 1000 nits) is displayed on an SDR display (peak luminance ≈ 200 nits), a *tone mapping operator (TMO)* is applied to compress the HDR range into the SDR range. Fuchsia's approach:

1) Escher's color correction subpass applies a Reinhard or ACES filmic TMO (configurable by Scenic policy) as a GPU fragment shader pass.
2) The tone-mapped output is clamped to $[0, 1]$ in sRGB.
3) The display coordinator's CSC/gamma pipeline takes the sRGB output and applies the display's native gamma correction.

For HDR displays, the tone mapping subpass is skipped entirely; the PQ-encoded values are passed directly through Escher to the display coordinator's PQ degamma LUT.

XVI. Debugging and Observability

A. fx trace and Perfetto

Fuchsia's tracing infrastructure is based on **Perfetto**. The `fx trace` command starts a system-wide trace session and records events from all participating components. For graphics debugging:

Listing 17. Capturing a graphics trace

```
fx trace --categories=gfx,input,magma,display \
         --duration=5 --output=trace.fxt
fx trace convert trace.fxt  # -> Perfetto JSON
# Open in ui.perfetto.dev
```

Key trace categories:

- **gfx**: `Present()`, aggregation, `ApplyConfig()`, `OnVsync()`, and jank events from Scenic and Flatland.
- **magma**: GPU command submission, semaphore waits, context switches, GPU hang detection.
- **display**: Display coordinator layer updates, vsync timestamps, capture events.
- **input**: Input Pipeline event routing, hit-test results, touch delivery latency.
- **memory**: VMO allocations, sysmem collection lifecycle, memory pressure events.

Each trace event has a monotonic timestamp from `zx_clock_get_monotonic()`, enabling precise cross-component latency measurement. The GPU submit→vsync latency (frame pipeline depth) is visible as the gap between the "ExecuteCommandBuffers" event and the "OnVsync" event in the `magma` and `gfx` tracks.

B. fx sysmem dump

`fx sysmem dump` prints a live snapshot of all active buffer collections:

Listing 18. Sample sysmem dump output

```
BufferCollection[0x4a2b]:
  name: "scenic-flatland-0" priority=10
  client: "scenic.cm" id=0x1
  heap: SYSTEM_RAM  contiguous: false
  buffers: 3  size: 7680000 bytes each
  format: R8G8B8A8  1920x1080 stride=7680
  refs: cpu_map=1 gpu_map=1 display_pin=1
```

The `refs` line shows how many active references of each type prevent deallocation. This is the primary tool for diagnosing buffer leaks. A buffer stuck with `display_pin=1` after a layer is removed indicates a missing `zx_pmt_unpin()` call in the display coordinator driver.

C. GPU Debugging: magma-inspect

The MSD exposes an **Inspect** vnode at `/dev/diagnostics/class/gpu/000`. The `fx inspect` command reads it:

Listing 19. Magma inspect output (Intel MSD)

```
gpu:
  device_id: 0x3e92  (Intel UHD 630)
  active_connections: 3
  pending_command_buffers: 7
  gpu_fault_count: 0
  last_reset_timestamp_ns: 0
  ring_head: 0x1a40  ring_tail: 0x1b80
  exec_count: 142857
  hw_context_count: 3
```

`ring_head` and `ring_tail` discrepancies indicate a stalled GPU (ring tail advanced but head not). `gpu_fault_count` increments on each GPU page fault; a non-zero value after a test run indicates a driver bug or incorrect GPU VA management.

D. Display Coordinator Debugging

The display coordinator exposes its state via `fuchsia.hardware.display/Info` and Inspect. The useful fields:

- **active_config_stamp**: which `ConfigStamp` is currently on screen.
- **vsync_count**: monotonically increasing vsync counter; gaps indicate dropped vsyncs (display controller overrun).
- **layer_count**: number of active hardware layers; exceeding the hardware plane count forces GPU compositing.
- **capture_in_progress**: whether a screenshot DMA is active.

E. GPU Timeline Tracing

Magma emits Perfetto trace events for the GPU command timeline. The MSD registers a `fuchsia.tracing.provider/Registry` provider and emits `trace_async_begin()` / `trace_async_end()` pairs around command buffer execution. Each pair is tagged with the

`context_id` and a monotonically increasing sequence number, so the Perfetto viewer can draw per-context GPU lanes in the trace.

To correlate CPU and GPU timelines, Magma uses the GPU's hardware timestamp counter. At initialization, the MSD reads the GPU timer frequency from a hardware register (`TIMESTAMP_FREQUENCY` on Intel, `JS_TIMESTAMP` on Mali) and records the CPU time (`zx_clock_get_monotonic()`) and GPU timer count simultaneously. This pair calibrates the GPU-to-CPU time conversion; subsequent GPU timestamps are converted and recorded as Zircon monotonic time in the trace, enabling direct CPU/GPU comparison in the Perfetto UI.

F. Flatland Debug Overlay

Scenic includes a built-in debug overlay (`-present_frame_stats`) that renders a real-time histogram of frame latencies and jank events as an overlay on the display. It is implemented as a Flatland session with a higher-priority Z-order that draws directly into a hardware overlay plane, so it adds zero GPU overhead to the application under test.

G. Key Zircon Syscalls for Graphics

Syscall	Graphics Use
`zx_vmo_create()`	Allocate GPU buffer
`zx_vmo_create_child()`	sysmem VMO hierarchy
`zx_vmar_map()`	CPU-map GPU buffer
`zx_cache_flush()`	GPU coherency on ARM
`zx_bti_pin()`	IOMMU pin for DMA
`zx_pmt_unpin()`	Release DMA mapping
`zx_event_create()`	Magma semaphore
`zx_object_wait_one()`	Wait on fence event
`zx_channel_write()`	Send FIDL message
`zx_channel_read()`	Receive FIDL reply
`zx_interrupt_wait()`	GPU IRQ handler
`zx_port_wait()`	Scenic async multiplex

H. Vulkan Validation Layers

Fuchsia supports Vulkan validation layers via the standard Vulkan loader mechanism. The layers are loaded as `.so` files from `/data/vulkan/implicit_layer.d/`. For Fuchsia-specific validation:

- **VK_LAYER_FUCHSIA_imagepipe_swapchain_fb**: validates imagepipe surface creation and swapchain teardown.
- **VK_LAYER_KHRONOS_validation**: standard Khronos validation; catches incorrect sysmem collection usage (e.g., binding a VMO from a wrong collection index).

Validation layer output is routed to Fuchsia's structured logging via `fuchsia.logger/LogSink`, viewable with `fx log -filter Vulkan`.

XVII. Vendor-Specific MSD Notes

A. ARM Mali MSD

The Mali MSD (`src/graphics/drivers/msd-arm-mali/`) targets the Bifrost and Valhall GPU families found on Amlogic SoCs used in Fuchsia reference hardware (Vim3, Sherlock). Key implementation details:

- **Job chains**: Mali command submission uses a linked-list of "atoms" (Job Descriptors). The MSD writes atom structs into a VMO and programs the Job Manager MMIO registers (`JM_ATOM_HEAD/TAIL`) to start execution.
- **ASID management**: Each Magma connection gets a unique ASID (1–255). The Mali MMU has one L1/L2 page table per ASID; the MSD programs `AS_TRANSTAB` and `AS_MEMATTR` for each active connection.
- **Protected mode**: Mali Bifrost supports a protected execution mode (PE mode) where the GPU refuses to read non-protected VMOs. The MSD switches into PE mode when the ICD binds protected sysmem memory, and back to normal mode on context switch. PE mode transitions are serialized with a GPU cache flush.
- **Power management**: The MSD integrates with Fuchsia's power framework (`fuchsia.power/ActivityGovernor`). The GPU power domain is asserted before any job chain is submitted and released after a 50ms idle timeout, using `MaliPowerManager` inside the MSD.

B. Intel GPU MSD

The Intel MSD (`src/graphics/drivers/msd-intel-gen/`) targets Gen9+ integrated GPUs (Kaby Lake, Coffee Lake) found in Chromebook-derived Fuchsia hardware:

- **Execlists**: Intel Gen9+ uses an Execlist submission mechanism. Command buffers are submitted via the `ELSP` (Execlist Submit Port) register. The MSD formats a 64-byte Execlist Descriptor pointing to a per-context LRCA (Logical Ring Context Area) VMO.
- **PPGTT**: Each Magma connection gets a 4-level per-process page table (PPGTT). The MSD allocates a page directory from a VMO pool and programs the per-context `PDP` registers in the LRCA.
- **Semaphores via MI_SEMAPHORE_WAIT**: Inter-context synchronization is implemented by writing `MI_SEMAPHORE_WAIT` instructions into the ring before the batch buffer pointer. The wait polls a GPU-visible memory location (a 4-byte slot in a "semaphore VMO") that the signaling context writes with `MI_STORE_DATA_IMM`.
- **Render and Blitter engines**: The Intel GPU has separate Render (3D/compute) and Blitter (BLT) command streamers. Magma exposes them as separate GPU contexts; the ICD can use the Blitter for async buffer copies without blocking the Render engine.

C. Goldfish (Emulator) MSD

The Goldfish MSD (`src/graphics/drivers/msd-virtio-gpu/`) implements the VirtIO GPU protocol for Fuchsia running in QEMU/KVM or Crosvm. Command buffers are passed as VirtIO descriptor chains to the host hypervisor, which renders them using the host's OpenGL/Vulkan. Goldfish uses the `GOLDFISH_DEVICE_LOCAL` sysmem heap, which allocates "guest VRAM". This memory is visible to both host and guest via a VirtIO BAR mapping. Zero-copy scanout is achieved by mapping the guest VRAM directly to the host's EGL surface.

XVIII. Putting It Together: Pixel Lifecycle

This section traces a single rendered game frame from application render call to photon emission, integrating all the preceding components.

A. Initialization (One-Time Setup)

Before the first frame, the application performs a one-time setup:

1) **Vulkan device creation**: `vkCreateInstance()` loads `libvulkan.so` (loader). The loader calls `GetIcdList()` on the MSD to find `libvulkan_vendor.so`. The ICD calls `Connect2()` on the MSD, obtaining Primary and Notification channels. `vkCreateDevice()` creates a Magma GPU context.
2) **Flatland session**: App opens `fuchsia.ui.composition/Flatland` via its component's incoming directory. Creates a root Transform. Opens `fuchsia.ui.composition/Allocator` for buffer collection registration.
3) **Buffer collection**: App calls `zx_channel_create()` to get a sysmem allocator channel. Creates a shared collection; duplicates tokens for the ICD (via `VK_FUCHSIA_buffer_collection`), for Flatland (via `Allocator/RegisterBufferCollection`), and for the display coordinator (via Flatland's internal sysmem call). sysmem waits for all three to call `SetConstraints()`.
4) **Constraint intersection**: sysmem intersects all three constraint sets. For a typical UI app: ICD wants RGBA8 or BGRA32 with `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`; Flatland wants linear or tiled with display scanout usage; display coordinator wants linear BGRA32. Result: linear BGRA32.
5) **VMO allocation**: sysmem allocates 3 VMOs (triple buffer), each $1920 \times 1080 \times 4 = 8{,}294{,}400$ bytes, from `SYSTEM_RAM` heap (or `CONTIGUOUS` if display requires). Returns VMOs to all participants.
6) **VkImage binding**: App calls `vkCreateImage()` with `VkBufferCollectionImageCreateInfoFUCHSIA` for each VMO index. ICD maps the VMO into the GPU address space via Magma `ImportBuffer()` + `MapBuffer()`. Display coordinator imports the VMOs via `ImportImage()` (called internally by Flatland).
7) **Flatland image registration**: App calls `Flatland/CreateImage()` with the `BufferCollectionImportToken` and declares the image size. Flatland assigns a `ContentId`. App calls `SetContent(transform_id, content_id)` to attach the image to a scene graph transform.

B. Per-Frame Render Loop

A concrete walk-through of a game frame:

1) **Init**: App connects to sysmem via `Allocator/RegisterBufferCollection`. sysmem intersects App + ICD + Flatland + Display constraints → negotiates BGRA32 linear VMOs. ICD binds VMOs as `VkImage` objects (swapchain).
2) **Render**: `vkAcquireNextImage()` returns when release fence from prior frame is signaled. App submits Vulkan draw commands; ICD packs a command buffer into a VMO, imports it via Magma `ImportBuffer()` and calls `ExecuteCommandBuffers()` with wait/signal semaphore IDs. MSD writes the command buffer pointer to the GPU ring buffer. GPU renders; completion interrupt fires; MSD signals the Vulkan completion semaphore event.
3) **Present to Flatland**: `vkQueuePresentKHR()` sends the image handle via imagepipe FIDL to Flatland. Flatland client also calls `SetContent()` + `Present()` with the GPU completion event as an acquire fence.
4) **Aggregation**: At frame deadline, FrameScheduler wakes. UberStructSystem snapshots all sessions with signaled acquire fences. Engine flattens scene graphs; occlusion culling removes hidden rects using clip bounds.
5) **Compositing decision**: DisplayCompositor calls `CheckConfig()` on the coordinator. If hardware can handle the layer stack (planes available, formats compatible, no unsupported blend modes), proceed to direct scanout. Otherwise: VkRenderer invokes Escher's RectangleCompositor, which GPU-composites all layers into one framebuffer. That framebuffer becomes the sole display plane.
6) **Scanout**: `ApplyConfig(ConfigStamp)` commits to hardware. Display controller DMAs from the VMO, PIPE blends overlay planes, color pipeline applies degamma/CSC/gamma, encoder transmits HDMI/DP signal.
7) **Vsync**: `OnVsync(ConfigStamp)` fires. Scenic signals release fences. App recycles the buffer. `OnFramePresented()` informs the app of actual display time. FrameScheduler updates its IIR vsync prediction for the next cycle.

XIX. Starnix: Linux Compatibility Graphics

Fuchsia's **Starnix** component implements a Linux syscall compatibility layer, allowing unmodified Android/Linux binaries to run on Fuchsia. Graphics in Starnix follows a different path than the native Fuchsia graphics stack:

A. DRM/KMS Emulation

Starnix emulates the Linux `/dev/dri/card0` device node and responds to DRM ioctls. For GPU access:

1) The Linux app opens `/dev/dri/renderD128` and issues `DRM_IOCTL_GEM_CREATE`, `DRM_IOCTL_PRIME_HANDLE_TO_FD`, etc.
2) Starnix intercepts these ioctls and translates them to Magma API calls: `GEM_CREATE` → `zx_vmo_create()`, `PRIME_FD_TO_HANDLE` → `ImportBuffer()`.
3) The Linux app's Vulkan ICD (e.g., Mesa's Turnip or RADV) loads via the standard `LD_LIBRARY_PATH` mechanism inside the Starnix container.

B. Wayland and XWayland Compositing

For display output, Starnix runs a Wayland compositor bridge:

- A Wayland server socket is exposed inside the Starnix container (`/run/wayland-0`).
- Android apps using SurfaceFlinger are bridged via an `android-surfaceflinger-bridge` Starnix module that translates `BufferQueue` operations to Wayland `wl_surface` commits.
- The Wayland bridge holds a native Flatland session and presents each Wayland surface as a Flatland `Image` backed by a sysmem collection that both the Starnix container's GPU and the Fuchsia compositor can access.
- Zero-copy is achieved: the Linux app renders into a PRIME dma-buf, which is the same Zircon VMO shared with Flatland.

C. Vulkan over Starnix

Android Vulkan apps in Starnix use the standard Android Vulkan loader (`libvulkan.so` from the Android NDK). The loader queries `/dev/dri/renderD128` for ICDs. Starnix's DRM emulation responds with the Fuchsia ICD (`libvulkan_vendor.so`). The ICD communicates with the MSD via Magma as normal. From the GPU driver's perspective, the app is running natively on Fuchsia; the Starnix syscall translation layer is transparent to Magma.

The `VK_FUCHSIA_buffer_collection` extension is not available to Starnix apps (it requires Fuchsia-native sysmem integration). Instead, Starnix maps `EGL_ANDROID_get_native_client_buffer` to a sysmem collection internally. The Wayland bridge negotiates the buffer collection between the Android app (GPU) and Flatland (display).

D. OpenGL ES Compatibility

For Android apps using OpenGL ES (GLES), Starnix provides:

- **ANGLE (Almost Native Graphics Layer Engine)**: Google's GLES-over-Vulkan translation layer. ANGLE is compiled as a Starnix library; it receives GLES calls and translates them to Vulkan API calls which then go through the Fuchsia ICD.
- **EGL surface**: ANGLE creates an EGL surface backed by an imagepipe Vulkan surface (`VK_FUCHSIA_imagepipe_surface`), connecting the GLES render output to Flatland.
- The performance overhead of GLES→ANGLE→Vulkan→Magma→GPU is typically 5–15% compared to native Vulkan, acceptable for most UI apps but not for compute-intensive 3D games.

E. Video Decode Pipeline

Media playback in Fuchsia uses a hardware video decoder accessed via `fuchsia.media.StreamProcessor`. The decode pipeline interacts with sysmem:

1) The media player opens a `StreamProcessor` (the video decoder service). The decoder negotiates an output buffer collection with sysmem, requesting NV12 format in a protected or unprotected heap depending on content DRM flags.
2) The GPU (ICD) sets `VkImageConstraintsInfoFUCHSIA` for NV12 `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM` into the same collection.
3) Flatland registers the collection with its Allocator.
4) After `WaitForAllBuffersAllocated()`, the decoder writes decoded NV12 frames into the VMOs; the GPU reads them as Vulkan NV12 images for color conversion; Flatland schedules them for display.
5) For DRM content, the entire pipeline uses the `AMLOGIC_SECURE` heap: decoder writes into protected memory, display DMA reads from it directly, and the GPU never touches it. The CPU cannot access any frame pixels.

XX. Conclusion

Fuchsia's graphics stack is a clean decomposition of concerns across dedicated user-space services, communicating via typed FIDL channels with zero-copy shared VMOs. For hardware engineers, the mental model maps well onto familiar RTL concepts:

- **sysmem** = format negotiation / handshake protocol between IP blocks.
- **Magma MSD** = PCIe endpoint driver owning register map.
- **Display coordinator** = CRTC/plane register abstraction.
- **Flatland** = frame buffer management and scanout scheduling.
- **Acquire/release fences** = valid/ready handshake signals.
- **VMOs with PMTs** = DMA buffer descriptors with physical address locking.
- **Capability handles** = hardware bus grants that cannot be forged.
- **IOMMU BTI** = address-space sandboxing for DMA engines.

The architecture prioritizes *isolation* (a crashing app cannot corrupt the compositor), *hardware utilization* (display planes rather than GPU blending when possible), and *deterministic latency* (per-session dispatchers, explicit fence-based pipelining, IIR vsync prediction). These properties make it a compelling model for FPGA-hosted SoC designs where the display pipeline, GPU, and compositor each live in separate hardware blocks that must share frame buffers without CPU involvement.

A. Lessons for FPGA GPU Design

For an FPGA engineer implementing a GPU or display pipeline that must integrate with Fuchsia, the key design requirements are:

1) **IOMMU compliance**: Implement a PCIe ATS (Address Translation Service) endpoint or platform IOMMU subordinate port. The Zircon BTI mechanism pins physical pages; your DMA engine must operate on IOMMU-translated addresses, not physical addresses, to participate in Magma's address space sandboxing.
2) **Zircon interrupt**: Map your GPU completion interrupt to a Zircon interrupt object (`zx_interrupt_create()`). The MSD's IRQ thread waits on this object; your hardware must assert the interrupt line after each command buffer completes.
3) **sysmem heap**: Register a custom sysmem heap for your on-chip SRAM or HBM. Implement the `fuchsia.sysmem2/Heap` FIDL protocol to allocate VMOs from your physical memory region. sysmem will call your heap allocator when a participant requests your heap type.
4) **Display plane**: Implement the `fuchsia.hardware.display` FIDL protocol in your display controller driver. At minimum: `ImportImage()`, `CreateLayer()`, `SetLayerImage()`, `ApplyConfig()`, and `OnVsync()` events. Implement `CheckConfig()` to report which layer configurations your hardware supports.
5) **Command ring**: Implement a hardware command ring buffer compatible with Magma's `ExecuteCommandBuffers()` model: a VMO-backed ring with head/tail registers, interrupt on completion, per-context address space switching.
6) **Protected memory**: If your design requires DRM content protection, implement a memory partition (analogous to TZC-400) that allows DMA only from specific engine ports (display DMA, video decoder DMA) to the protected physical region.

The Fuchsia graphics stack rewards hardware that implements these primitives cleanly. In return it provides zero-copy buffer sharing, capability-safe process isolation, and a well-specified vsync-locked frame scheduling protocol with open-source compositor and driver reference implementations.

B. Summary Table: Component Responsibilities

Component	Owns	Key FIDL
Magma MSD	GPU registers, HW address space	`fuchsia.gpu.magma/Primary`
Magma ICD	SPIR-V compilation, cmd buffers	(in-process, no FIDL)
sysmem	Buffer allocation, format negotiation	`fuchsia.sysmem2/Allocator`
Display coord	CRTC, planes, encoder	`fuchsia.hardware.display/Coordinator`
Flatland	Scene graph, compositing decision	`fuchsia.ui.composition/Flatland`
Escher	Vulkan GPU compositing	(C++ library, no FIDL)
Input Pipeline	HID normalization, hit-test	`fuchsia.ui.pointer/TouchSource`

References

[1] Google, "Magma Design," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/graphics/magma/concepts/design

[2] Google, "sysmem Overview," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/graphics/sysmem/concepts/sysmem

[3] Google, "Flatland," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/ui/scenic/flatland

[4] Google, "Life of a Pixel," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/ui/scenic/life_of_a_pixel

[5] Google, "Escher," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/graphics/escher

[6] Google, "Display Hardware Concepts," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/drivers/driver_guides/display/hardware_concepts

[7] Google, "Frame Scheduling," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/graphics/scenic/concepts/frame_scheduling

[8] Google, "FIDL Overview," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/fidl/overview

[9] Google, "Zircon Kernel Objects," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/objects

[10] Google, "Driver Framework," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/drivers/driver_framework