# Android graphics HAL

Graphics

bookmark_border
Android Graphics HAL icon
The Android framework offers a variety of graphics rendering APIs for 2D and 3D that interact with manufacturer implementations of graphics drivers, so it is important to have a good understanding of how those APIs work at a higher level. This page introduces the graphics hardware abstraction layer (HAL) that those drivers are built on. Before continuing with this section, familiarize yourself with the following terms:

canvas (generic term), Canvas (API element)
A canvas is a drawing surface that handles compositing of the actual bits against a bitmap or a Surface object. The Canvas class has methods for standard computer drawing of bitmaps, lines, circles, rectangles, text, and so on, and is bound to a bitmap or surface. A canvas is the simplest, easiest way to draw 2D objects on the screen. The base class is Canvas.
drawable
A drawable is a compiled visual resource that can be used as a background, title, or other part of the screen. A drawable is typically loaded into another UI element, for example as a background image. A drawable can't receive events, but assigns various other properties such as state and scheduling, to enable subclasses such as animation objects or image libraries. Many drawable objects are loaded from drawable resource files — XML or bitmap files that describe the image. Drawable resources are compiled into subclasses of android.graphics.drawable. For more information about drawables and other resources, see App resources overview.
layout resource
A layout resource is a XML file that describes the layout of an activity screen. For more information, see Layout resource.
nine-patch (9-patch, NinePatch)
A nine-patch is a resizeable bitmap resource that can be used for backgrounds or other images on the device. For more information, see Nine-patch.
OpenGL ES
OpenGL ES is a cross-platform API for rendering 2D and 3D graphics. Android provides OpenGL ES libraries for hardware-accelerated 3D rendering. For 2D rendering, a canvas is the simpler option. OpenGL ES is available in the Android Native Development Kit (NDK). The android.opengl and javax.microedition.khronos.opengles packages expose OpenGL ES functionality.
surface (generic term), Surface (API element)
A surface represents a block of memory that gets composited to the screen. A surface holds a canvas for drawing, and provides various helper methods to draw layers and resize the Surface

object. Use the SurfaceView class instead of the Surface class directly.
surface view (generic term), SurfaceView (API element)
A surface view is a View object that wraps a Surface object for drawing, and exposes methods to specify its size and format dynamically. A surface view provides a way to draw independently of the UI thread for resource-intensive operations, such as games or camera previews, but it uses extra memory as a result. A surface view supports both canvas and OpenGL ES graphics. The base class for a SurfaceView object is SurfaceView.
theme
A theme is a set of properties, such as text size and background color, bundled together to define various default display settings. Android provides a few standard themes, listed in R.style and prefaced with Theme_.
view (generic term), View (API element)
A view draws a rectangular area on the screen and handles click, keystroke, and other interaction events. The View class is the base class for most layout components of an activity or dialog screen, such as text boxes and windows. A View object receives calls from its parent object (see ViewGroup) to draw itself, and informs its parent object about its preferred size and located, which might not be respected by the parent. For more information, see View.
view group (generic term), ViewGroup (API element)
A view group groups a set of child views. The view group is responsible for deciding where child views are positioned and how large they can be, as well as for calling each to draw themselves when appropriate. Some view groups are invisible and are for layout only, while others have an intrinsic UI, such as a scrolling list box. View groups are in the android.widget package, but extend the ViewGroup class.
view hierarchy
A view hieararchy is an arrangement of view and view group objects that defines the user interface for each component of an app. The hierarchy consists of view groups that contain one or more child views or view groups. You can obtain a visual representation of a view hierarchy for debugging and optimization by using the Hierarchy Viewer that is supplied with the Android SDK.
Vulkan
Vulkan is a low-overhead cross-platform API for high-performance 3D graphics.
widget
A widget is one of a set of fully implemented view subclasses that render form elements and other UI components, such as a text box or a popup menu. Because a widget is fully implemented, it handles measuring, drawing itself, and responding to screen events. Widgets are in the android.widget package.
window (generic term), Window (API element)
In an Android app, a window is an object derived from the Window abstract class that specifies the elements of a generic window, such as the look and feel, title bar text, and location and content of menus. Dialogs and activities use an implementation of the Window class to render a Window object. You don't need to implement the Window class or use windows in your app.
App developers draw images to the screen in three ways: with Canvas, OpenGL ES, or Vulkan.

This document introduces **Android Graphics** concepts from the **framework level** down to the **graphics HAL and drivers**. Here's a simplified breakdown to help you understand everything clearly:

# 🔷 What Is Android Graphics HAL?

The **Graphics HAL** is the layer that **connects Android's rendering APIs (Canvas, OpenGL ES, Vulkan)** to the **hardware-specific graphics drivers** provided by device manufacturers.

It ensures your app's drawing calls are rendered properly on the screen using the actual GPU.

---

# 🔧 Key Graphics Concepts in Android

## 1. Canvas

- **What**: A simple 2D drawing surface.
- **Use case**: Drawing lines, circles, text, and bitmaps.
- **API**: `android.graphics.Canvas`.
- **Backed by**: A `Bitmap` or a `Surface`.

## 2. Drawable

- **What**: A visual element like an image, shape, or animation.
- **Use case**: Backgrounds, icons, etc.
- **Loaded from**: XML or bitmap files (`res/drawable/`).

## 3. Layout Resource

- **What**: An XML file describing UI layout.
- **Example**: `activity_main.xml`.
- **Use case**: Positioning views and widgets on screen.

## 4. Nine-Patch

- **What**: A stretchable image format.
- **Use case**: Scalable UI backgrounds that adapt to screen size.
- **Tool**: `draw9patch` in SDK.

---

# 🔺 Rendering APIs

## 1. OpenGL ES

- **What**: A C/C++ based 2D/3D hardware-accelerated rendering API.

- **Use case**: Games, simulations, complex 3D UIs.
- **Android Support**: via Java (`android.opengl`) and NDK.

### 2. Vulkan

- **What**: A low-overhead, high-performance 3D rendering API.
- **Use case**: Advanced 3D graphics with fine control and multithreading.
- **More efficient** than OpenGL ES, but more complex.

---

## 🖼️ Surface and Views

### 1. Surface

- **What**: A block of memory where pixels are drawn before being shown.
- **Used by**: Camera preview, videos, games.
- **Backs**: `Canvas` or OpenGL drawing.

### 2. SurfaceView

- **What**: A `View` that wraps a `Surface`.
- **Benefit**: Allows drawing **outside UI thread**.
- **Use case**: High-performance rendering (e.g., media playback, games).

---

## 📐 Views and UI

### 1. View

- **What**: Base class for all UI components.
- **Handles**: Drawing, user interaction.

### 2. ViewGroup

- **What**: A container for multiple `View` or `ViewGroup` children.
- **Examples**: `LinearLayout`, `RelativeLayout`.

### 3. View Hierarchy

- **What**: A tree of `View` and `ViewGroup` objects.
- **Built from**: Layout XML or programmatically.

- **Debug Tool**: Android Studio's **Layout Inspector** or **Hierarchy Viewer**.

---

## 🧱 Other Terms

**• Theme: A set of default styling properties (colors, fonts, etc.).**

**• Widget: A fully implemented UI component (Button, TextView, etc.).**

**• Window: Top-level UI container (used internally by Android).**

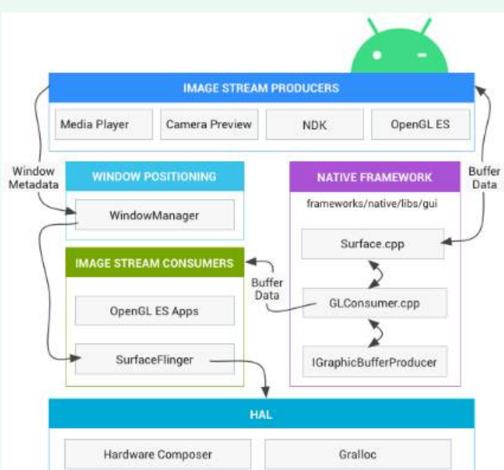
---

## ✅ Summary: 3 Ways to Render in Android

| Rendering Method | Use Case | Pros | Cons |
|---|---|---|---|
| **Canvas** | 2D drawing, simple UIs | Easy to use, Java API | Limited performance |
| **OpenGL ES** | Games, 3D apps | Cross-platform, hardware-accelerated | More complex, C++/NDK usage |
| **Vulkan** | Advanced 3D, low-level control | High performance, multithreaded | Very complex API |

---

Would you like to dive into the **Graphics HAL implementation**, **SurfaceFlinger**, or **how GPU buffers are used** behind the scenes?




Android graphics components
No matter what rendering API developers use, everything is rendered onto a surface. The surface represents the producer side of a buffer queue that is often consumed by SurfaceFlinger. Every window that is created on the Android platform is backed by a surface. All of the visible surfaces rendered are composited onto the display by SurfaceFlinger.

The following diagram shows how the key components work together:

Figure 1. How surfaces are rendered.

The main components are described in the following sections.

Image stream producers
An image stream producer can be anything that produces graphic buffers for consumption. Examples include OpenGL ES, Canvas 2D, and mediaserver video decoders.

Image stream consumers
The most common consumer of image streams is SurfaceFlinger, the system service that consumes the currently visible surfaces and composites them onto the display using information provided by the Window Manager. SurfaceFlinger is the only service that can modify the content of the display. SurfaceFlinger uses OpenGL and the Hardware Composer (HWC) to compose a group of surfaces.

Other OpenGL ES apps can consume image streams as well, such as the camera app consuming a camera preview image stream. Non-GL apps can be consumers too, for example the ImageReader class.

Hardware Composer
The hardware abstraction for the display subsystem. SurfaceFlinger can delegate certain composition work to the HWC to offload work from OpenGL and the GPU. SurfaceFlinger acts as just another OpenGL ES client. So when SurfaceFlinger is actively compositing one buffer or two into a third, for instance, it's using OpenGL ES. This makes compositing lower power than having the GPU conduct all computation.

The Hardware Composer HAL conducts the other half of the work and is the central point for all Android graphics rendering. The HWC must support events, one of which is VSync (another is hotplug for plug-and-play HDMI support).

Thanks! You've provided a detailed image and explanation of how **Android's graphics pipeline** works. Let's break it down **step-by-step** with clarity, referring to the diagram you posted and expanding on the roles of each component:

---

# 🧠 Core Idea

Everything visible on an Android device is drawn to a **`Surface`**, which is a **buffer queue** used between a **producer** and a **consumer**.

- **Producers** generate frames.

- **Consumers** read and display or process those frames.
- All surfaces are ultimately composited by **SurfaceFlinger** and sent to the **display hardware** via the **Hardware Composer HAL (HWC)**.

---

## 🧵 1. Image Stream Producers (Top-Blue)

These are components that *generate frames or graphic buffers*:

- **Media Player** – video frames
- **Camera Preview** – camera output
- **NDK** – native apps with custom rendering
- **OpenGL ES** – 3D apps, games

➡️ These producers write into a **`Surface`**, which is a shared memory queue.

---

## 2. Window Positioning (Mid-Light-Blue)

- **WindowManager** determines **where each surface appears on screen**.
- Sends **window metadata** like size, Z-order, transparency to SurfaceFlinger.

---

## 🧰 3. Native Framework (Mid-Purple)

These C++ framework components live in AOSP:

| Component | Role |
|---|---|
| `Surface.cpp` | App-facing interface for writing to a buffer |
| `GLConsumer.cpp` | Reads buffers and makes them available to GL (or SurfaceFlinger) |
| `IGraphicBufferProducer` | Interface between producer (app) and consumer (GL, SurfaceFlinger) |

> A **`Surface`** is typically backed by a **`GraphicBuffer`** from Gralloc (GPU memory).

---

## 🖼️ 4. Image Stream Consumers (Mid-Green)

These are consumers of frames from **`Surface`**:

- **SurfaceFlinger** – *Composites all visible surfaces* into one final frame for display.
- **OpenGL ES Apps** – Can be consumers (e.g. for camera texture).
- **ImageReader (not shown)** – Can also read from Surface buffer queue.

# ⚙️ 5. HAL Layer (Bottom-Cyan)

- **Gralloc** (Graphics Allocator):
  - Allocates memory buffers (graphic buffers).
  - Provides access to shared GPU/CPU memory.
- **Hardware Composer (HWC)**:
  - Aids SurfaceFlinger in composition.
  - Can **offload layers** directly to display hardware (power-efficient).
  - Supports events like **VSync** and **Hotplug**.

  Think of **HWC** as the final **"conductor"** that helps draw final output on the screen, skipping GPU if possible.

---

# 🔁 Flow Example: Camera Preview

1. Camera HAL writes frames to a `Surface`.
2. The `Surface` acts as a producer queue.
3. The camera app (OpenGL ES) reads the buffer as a `GLConsumer`.
4. SurfaceFlinger composites the frame with other surfaces (like buttons).
5. HWC shows the final output to the screen.

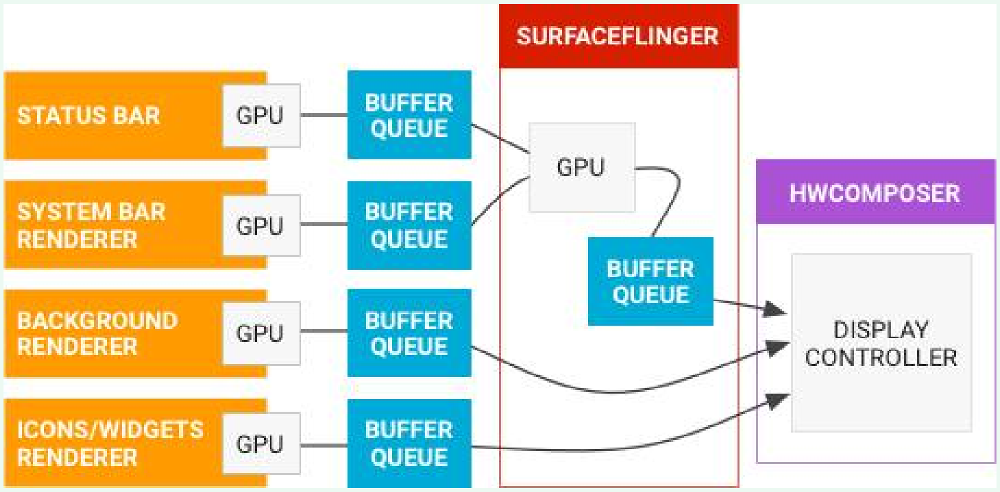
---

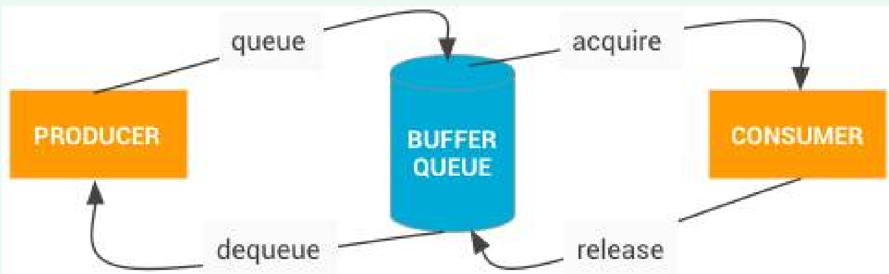
# ✅ Summary Table

| Component | Role |
|---|---|
| **Surface** | Buffer queue between producer and consumer |
| **SurfaceFlinger** | System compositor; final authority on what's shown |
| **WindowManager** | Layout and window Z-ordering |
| **HWC** | HAL for efficient composition and display |
| **Gralloc** | Allocates GPU-friendly memory buffers |
| **IGraphicBufferProducer/Consumer** | Interfaces managing buffer flow |

---

If you'd like, I can walk you through how this all works **in a live app** (like a camera or video player), or how to **trace buffer flow** using `dumpsys SurfaceFlinger`, `systrace`, or `perfetto`. Would that help?




Gralloc
The graphics memory allocator (Gralloc) is needed to allocate memory requested by image producers. For details, see BufferQueue and Gralloc.

Data flow
The following diagram depicts the Android graphics pipeline:

Figure 2. Graphic data flow through Android.

The objects on the left are renderers producing graphics buffers, such as the home screen, status bar, and system UI. SurfaceFlinger is the compositor and the HWC is the composer.

BufferQueue
BufferQueues provide the glue between the Android graphics components. These are a pair of queues that mediate the constant cycle of buffers from the producer to the consumer. After the producers hand off their buffers, SurfaceFlinger is responsible for compositing everything onto the display.

The following diagram illustrates the BufferQueue communication process:

Figure 3. BufferQueue communication process.

BufferQueue contains the logic that ties image stream producers and image stream consumers together. Some examples of image producers are the camera previews produced by the camera HAL or OpenGL ES games. Some examples of image consumers are SurfaceFlinger or another app that displays an OpenGL ES stream, such as the camera app displaying the camera viewfinder.

BufferQueue is a data structure that combines a buffer pool with a queue and uses Binder IPC to pass buffers between processes. The producer interface, or what you pass to somebody who wants to generate graphic buffers, is IGraphicBufferProducer (part of SurfaceTexture). BufferQueue is often used to render to a Surface and consume with a GL Consumer, among other tasks.

BufferQueue can operate in three different modes:

synchronous-like mode
BufferQueue by default operates in a synchronous-like mode, in which every buffer that comes in from the producer goes out at the consumer. No buffer is ever discarded in this mode. And if the producer is too fast and creates buffers faster than they are being drained, it blocks and waits for free buffers.
non-blocking mode
BufferQueue can also operate in a non-blocking mode where it generates an error rather than waiting for a buffer in those cases. No buffer is ever discarded in this mode either. This is useful for avoiding potential deadlocks in application software that might not understand the complex dependencies of the graphics framework.
discard mode
BufferQueue may be configured to discard old buffers rather than generate errors or wait. For instance, if conducting GL rendering to a texture view and drawing as quickly as possible, buffers must be dropped.
To conduct most of this work, SurfaceFlinger acts as just another OpenGL ES client. So when SurfaceFlinger is actively compositing one buffer or two into a third, for instance, it's using OpenGL ES.

The Hardware Composer HAL conducts the other half of the work. This HAL acts as the central point for all Android graphics rendering.

Perfect — with these diagrams and your notes, let me explain the **Android graphics pipeline** clearly and completely, from **renderers to the display**, with emphasis on **BufferQueue**, **Gralloc**, **SurfaceFlinger**, and **HWC**.

---

## 🧱 OVERALL STRUCTURE: Android Graphics Pipeline

We can split it into 5 main layers:

| Layer | Components |
|---|---|
| App/UI Layer | Status bar, System UI, Apps |
| GPU | Renders frame content to memory |
| BufferQueue | Transfers rendered frames to SurfaceFlinger |

| Layer | Components |
|---|---|
| SurfaceFlinger | Composites layers together |
| HWC + Display | Shows final frame on screen |

---

# 🔁 BUFFERQUEUE: Central to Graphics Communication

## 📌 What is it?

`BufferQueue` is the **producer-consumer** bridge between components.

- Producers: Apps, camera, video decoders, etc.
- Consumers: SurfaceFlinger, GL renderers, etc.

It supports **three key steps**:

1. `dequeue()` – producer asks for a free buffer
2. `queue()` – producer submits a filled buffer
3. `acquire()` – consumer grabs the buffer for use
4. `release()` – consumer returns it for reuse

## 📷 Real Example:

- Camera HAL writes preview frames into BufferQueue.
- Camera App (GL Consumer) displays them in a `SurfaceView`.

---

# 📦 GRALLOC: GPU-Friendly Memory Allocation

Gralloc = **Graphics Allocator HAL**

- Allocates memory buffers for use by GPU and CPU.
- Ensures memory is suitable for fast access by hardware (e.g. tiled, aligned).
- Works alongside BufferQueue — the memory allocated by Gralloc is used in the buffers inside BufferQueue.

So:

- When you call `dequeueBuffer()` → Gralloc gives you memory.
- When you `queueBuffer()` → SurfaceFlinger uses it or passes it to HWC.

---

# 🧩 SURFACEFLINGER: The Central Compositor

SurfaceFlinger is:

- A system service that **composites all visible layers** from apps and the system UI.
- The only service allowed to directly modify what's shown on screen.

### It handles:

- Compositing layers using OpenGL ES or HWC.
- Managing VSync timing.
- Communicating with HWC to optimize performance.

It reads buffers from all BufferQueues and combines them into a **single output frame**.

---

# ⚙️ HARDWARE COMPOSER (HWC)

HWC = HAL module for display hardware.

SurfaceFlinger asks HWC:

- Can you handle this layer (e.g., plane composition)?
- Can you blend these 2 layers in hardware?

If HWC says "yes":

- SurfaceFlinger offloads work (saves GPU power).
  If HWC says "no":
- SurfaceFlinger composites using OpenGL ES.

Then:

- Final buffer is sent to **Display Controller** by HWC.

---

# 🔁 FLOW EXPLAINED (With Diagrams You Posted)

## 1. App/UI Rendering

- Status bar, system UI, etc. render frames via GPU.
- Output goes into **BufferQueue** (one per component).

## 2. SurfaceFlinger

- Collects all these BufferQueues.
- For each frame:
  - Reads available buffers.
  - Composites using GPU or HWC.
  - Sends result to another BufferQueue.

### 3. HWC

- Gets composed buffer from SurfaceFlinger.
- Displays it via **Display Controller**.

---

## 🔧 BUFFERQUEUE MODES

| Mode | Behavior | Use Case Example |
|---|---|---|
| **Synchronous** | Producer blocks if no free buffer | General UI rendering |
| **Non-blocking** | Producer gets error if no free buffer | Deadlock-sensitive apps |
| **Discard mode** | Older buffers are dropped if no free buffer | Fast camera preview or streaming |

---

## 🧠 Summary

| Component | Role |
|---|---|
| **`Gralloc`** | Allocates GPU-friendly memory |
| **`BufferQueue`** | Connects producers and consumers of graphic buffers |
| **`SurfaceFlinger`** | Composites layers and communicates with HWC |
| **`HWC`** | Handles hardware overlays and final frame composition |
| **`Display Controller`** | Shows final frame on screen |

---

If you'd like, I can walk you through:

- A specific BufferQueue example (**`SurfaceView`** with camera),
- How to trace rendering using **`dumpsys SurfaceFlinger`**,
- Or internal AOSP code like **`BufferQueueProducer`**, **`SurfaceFlinger::doComposition`**.

Would you like that?

Surface vs SurfaceVIew

Great question. Let's clarify the difference between **`Surface`** and **`SurfaceView`**, two essential but **very different components** in Android graphics.

---

## 🔹 1. What is a `Surface`?

A **`Surface`** is a **low-level drawing target** — a block of memory (usually backed by Gralloc) that apps and system services use to render frames. It represents the **producer side** of a **BufferQueue**.

### ✅ Key Points:

- Not a UI widget — you don't "see" a **`Surface`** directly.
- It is used behind-the-scenes by **`SurfaceView`**, **`TextureView`**, **`SurfaceHolder`**, camera previews, video decoders, etc.
- Used with:
  - **Canvas** (via **`lockCanvas()`** / **`unlockCanvasAndPost()`**)
  - **OpenGL ES** (via EGL surface binding)
  - **NDK** (via **`ANativeWindow`**)

### 🔧 Example:

```java
Surface surface = surfaceHolder.getSurface();
Canvas canvas = surface.lockCanvas(null);
// draw on canvas
surface.unlockCanvasAndPost(canvas);
```

---

## 🔷 2. What is a `SurfaceView`?

A **`SurfaceView`** is a **UI component** (a subclass of **`View`**) that **contains and manages a `Surface`** for you. It allows rendering **in a separate thread** or **native code**, outside the normal UI **`View`** rendering pipeline.

### ✅ Key Points:

- Visible part of the layout, placed like a normal **`View`**.
- Provides access to the underlying **`Surface`** through a **`SurfaceHolder`**.
- Commonly used for:

  - Video playback
  - Camera preview
  - Custom OpenGL rendering (e.g., games)
- **Renders in a separate layer** (Z-order) — can appear *above or below* other views.

---

## 🔄 Relationship: `SurfaceView` wraps a `Surface`

- When you use **`SurfaceView`**, Android internally creates a **`Surface`**.
- You get that **`Surface`** via a **`SurfaceHolder`**:

```java
surfaceView.getHolder().getSurface();
```

---

## 🆚 Surface vs SurfaceView – Comparison Table

| Feature | `Surface` | `SurfaceView` |
|---|---|---|
| Type | Low-level rendering target | UI widget (View) with a **`Surface`** inside |
| Role | Used for rendering with GL/Canvas | Provides a place to show **`Surface`** output |
| Rendering thread | App-controlled | App-controlled (but outside UI thread) |
| Belongs to View system? | ❌ No | ✅ Yes |
| Drawing API | Canvas, OpenGL, Vulkan | Must access via **`SurfaceHolder`** |
| Used for | Internal rendering target | Game view, camera preview, video |
| Can be in layout XML? | ❌ No | ✅ Yes (**`<SurfaceView />`**) |

---

## 📌 When to Use What?

| Use Case | Use This |
|---|---|
| You need a place to draw with GL | **`SurfaceView`** |
| You have an external **`Surface`** from Camera/MediaCodec | **`Surface`** |
| You want low-latency drawing | **`SurfaceView`** |
| You want to mix UI + Video | **`TextureView`** *(better layering control)* |

---

## 🔍 Bonus: `SurfaceView` vs `TextureView`?

| Feature | `SurfaceView` | `TextureView` |
|---|---|---|
| Backed by | Hardware overlay (**`Surface`**) | OpenGL texture |
| UI compositing | Drawn in a separate layer | Composited with other views |
| Z-order issues | Yes (can appear above/below views) | No (normal view stacking) |
| Transformations | Not supported (scale/rotate) | Fully supported (matrix, alpha) |
| Latency | Lower | Slightly higher |

---

Would you like a **working sample app** that uses **`SurfaceView`** and draws with Canvas or OpenGL ES?

BufferQueue
what are the content in this

## 🔄 `BufferQueue` in Android: Core Component of Graphics Buffer Management

**`BufferQueue`** is a central mechanism in Android's graphics architecture, enabling **buffer exchange** between a **producer** (e.g. camera, OpenGL, video decoder) and a **consumer** (e.g. SurfaceFlinger, GPU, MediaCodec, etc.).

---

## 🧱 What is inside a `BufferQueue`?

At a high level, it manages a queue of **graphic buffers** shared between producer and consumer. Here's what it contains:

---

### 📦 1. Graphic Buffers

- Actual memory blocks (gralloc buffers) that store pixel data (RGBA/YUV etc).
- Managed via **`GraphicBuffer`** objects.
- Shared between producer and consumer using **ashmem** or **ion/dma-buf** (zero-copy).

---

### 🔄 2. Buffer Slots

- Fixed-size circular array of slots (usually 64).
- Each slot can hold one **`GraphicBuffer`**.
- Status: FREE, DEQUEUED, QUEUED, ACQUIRED
- Only one producer and one consumer allowed per BufferQueue instance.

---

### 📥 3. Producer Interface (`IGraphicBufferProducer`)

- Exposed to producers (e.g. camera, OpenGL ES, MediaCodec).
- Methods:
  - **`dequeueBuffer()`** – ask for an empty slot
  - **`queueBuffer()`** – submit a filled buffer

---

### 📤 4. Consumer Interface (`IGraphicBufferConsumer`)

- Used by consumers (e.g. SurfaceFlinger, GPU renderer).
- Methods:
  - **`acquireBuffer()`** – take a ready buffer
  - **`releaseBuffer()`** – return buffer after display/use

---

### 🔁 5. Synchronization Primitives

- **Fences (sync fences):**
  - File descriptors (FDs) used to signal when GPU/CPU operations complete.
  - **`queueBuffer()`** and **`acquireBuffer()`** exchange sync fences for proper sequencing.

---

### ⏱️ 6. Timestamps and Metadata

- Buffer timestamps (used for VSYNC synchronization).
- Transform flags (rotation, scaling).
- Crop/region data.
- Scaling mode, dataspace (colorspace), etc.

---

### 👥 7. Connection Info

- Producer/consumer connection status.
- Ownership rules: who owns the slot at each point.

---

### 🧠 8. State Machine

- Internal logic tracks:
  - Which slot is filled

- Which buffer is ready for display
- When to return or reallocate

## ✅ Summary Table

| Component | Description |
|---|---|
| `GraphicBuffers` | Actual frame/pixel memory blocks |
| `Slots` | Fixed-size array holding buffers (64 max) |
| `Producer API` | Used by app side (Surface/SurfaceView) |
| `Consumer API` | Used by SurfaceFlinger, OpenGL, etc. |
| `Sync Fences` | Ensure proper GPU/CPU scheduling |
| `Metadata` | Timestamp, crop, transform, scaling |
| `State Machine` | Manages flow and ownership |

Would you like a **flow diagram** to visualize how data moves through the BufferQueue between `SurfaceView` and `SurfaceFlinger`?

How many bufferqueue will be created?

Great question. Let's break this down clearly:

# ✅ How many `BufferQueue` instances get created?

### 🔹 One BufferQueue per Surface

In Android, **each `Surface` or `SurfaceView` creates one `BufferQueue`.**

So:

| Component | `BufferQueue?` |
|---|---|
| `SurfaceView` | ✅ Yes, 1 |
| `SurfaceTexture` | ✅ Yes, 1 |
| `ImageReader` | ✅ Yes, 1 |
| `MediaCodec` (input/output) | ✅ Yes, 1 per port |
| `GLSurfaceView` | ✅ Yes, 1 |
| `Camera HAL Stream` | ✅ Yes, per stream |
| `SurfaceFlinger` | ❌ Uses BufferQueues as **consumer** only |

| Component | BufferQueue? |
|---|---|
| **App Window** (via SurfaceControl) | ✅ Yes (backed by Surface) |

---

# 🔄 Producer ↔ Consumer: One Pair per BufferQueue

A single **`BufferQueue`** has:

- **One producer** (e.g. app, camera, OpenGL)
- **One consumer** (e.g. SurfaceFlinger, GPU)

If you have **two SurfaceViews on screen**, there will be **two BufferQueues** – one for each.

---

# 🔍 Typical Scenarios:

## 1. App UI with SurfaceView (e.g., video player):

- 1 BufferQueue for the **`SurfaceView`**.
- Producer: video decoder (**`MediaCodec`**)
- Consumer: **`SurfaceFlinger`** for display

## 2. Camera Preview:

- 1 BufferQueue for the preview Surface.
- Producer: camera HAL
- Consumer: **`SurfaceView`** or **`ImageReader`**

## 3. Multiple Surfaces in an app:

- Each creates its own **`BufferQueue`**.

---

# 🧠 Conclusion:

- ✅ **1 `BufferQueue` per `Surface`**
- ⚠️ **Multiple BufferQueues** if you use multiple surfaces
- 🔁 Each BufferQueue is point-to-point: one producer, one consumer

---

If you'd like, I can show a diagram with multiple surfaces and how their BufferQueues connect to producers/consumers. Would that help?

relation of bufferqueue with surface

Great — let's explain the **relationship between `BufferQueue` and `Surface`** step-by-step in **simple terms**, with **clear roles**:

---

## ✅ 1. What is a Surface?

A **`Surface`** in Android is an **interface** for **drawing something that will be shown on screen**. You **don't draw directly** on the Surface — instead, you write to a **graphics buffer**, and the Surface handles sending it to the **display system**.

---

## ✅ 2. What is a BufferQueue?

A **`BufferQueue`** is a **FIFO (First In, First Out) queue** of **graphics buffers** that connects:

- A **producer** (who fills buffers with image frames)
- A **consumer** (who displays or processes those buffers)

---

## 📌 3. How Surface and BufferQueue are related

```text
App/GL/Camera
    ↓
 [Producer]
    ↓
[BufferQueue]
    ↓
[Consumer (SurfaceFlinger/GPU/etc)]
```

- **Every Surface internally contains a BufferQueue.**
- The app (or MediaCodec, OpenGL, etc.) is the **producer**.
- SurfaceFlinger or GPU is typically the **consumer**.

---

## 🔧 4. Internals: What happens when you create a Surface?

When you create a **`Surface`**:

1. Android creates a **`BufferQueue`**.

2. The **producer side** is given to the app (or GL, or decoder).
3. The **consumer side** is kept by SurfaceFlinger (for display), or ImageReader, etc.

🧩 In code:

```java
Surface surface = surfaceView.getHolder().getSurface();
```

This gives the app the **producer** side of the BufferQueue (via **`IGraphicBufferProducer`**).

---

## 🔁 5. Drawing Flow (Simplified):

Let's say you have a SurfaceView showing camera preview:

1. **Camera HAL** is the **producer**, and it writes image frames to the BufferQueue.
2. **SurfaceFlinger** is the **consumer**, and it reads frames from the BufferQueue to show on screen.

So:

```scss
Camera HAL (producer)
     ↓
  BufferQueue (in Surface)
     ↓
  SurfaceFlinger (consumer)
```

---

## 🔍 6. Who Owns Buffers in BufferQueue?

There are 3 buffer states:

- **Dequeued**: Producer has the buffer
- **Queued**: In the queue waiting for consumer
- **Acquired**: Consumer is using the buffer

---

## 🧠 Summary:

| Concept | Role |
|---|---|
| **`Surface`** | Abstraction for display, wraps BufferQueue |
| **`BufferQueue`** | Queue of buffers between producer and consumer |
| Producer | App/GL/MediaCodec/Camera/etc. |

| Concept | Role |
|---|---|
| Consumer | SurfaceFlinger, GPU, ImageReader, etc. |
| Relationship | **Each Surface creates a BufferQueue** and provides producer access |

---

Let me know if you want a **visual diagram** or flowchart of this — I can make one for you.

confusing

No problem — let's make it **very simple** and visual.

---

## 🟩 1. Surface is just a wrapper for BufferQueue

Think of **`Surface`** like a **mailbox**.

Inside the mailbox is a **queue** (BufferQueue) where **images (frames)** are dropped by someone, and picked up by someone else.

---

## 🟦 2. BufferQueue = Two Ends

```scss
Producer (writes images)  →  BufferQueue  →  Consumer (reads images)
```

- **Producer**: Your app, Camera, OpenGL, or Video Decoder
- **Consumer**: SurfaceFlinger (draws on screen), GPU, or ImageReader

🧠 So, **Surface contains the BufferQueue**, and gives the **producer side** to whoever wants to draw.

---

## 🟨 3. Example: SurfaceView (showing camera)

```markdown
Camera HAL           →        BufferQueue       →     SurfaceFlinger
   (Producer)                    (Inside)               (Consumer)
                             Surface/SurfaceView
```

1. You create a **`SurfaceView`** → it creates a **`Surface`**
2. **`Surface`** sets up a **`BufferQueue`**

3. Camera HAL gets the **producer side**
4. Camera writes image → goes into BufferQueue
5. SurfaceFlinger reads image from BufferQueue → shows it on screen

---

## 🟥 Key Point

👉 **Every `Surface` has a BufferQueue.**

- Producer puts image buffers **into** the queue
- Consumer takes buffers **out** and shows them

---

## ✅ Simple Table:

| Term | What it does |
|---|---|
| `Surface` | Wraps a BufferQueue |
| `BufferQueue` | Connects Producer ↔ Consumer |
| `Producer` | App / Camera / OpenGL |
| `Consumer` | SurfaceFlinger / GPU / ImageReader |

---

Would you like a simple picture or flowchart to make it crystal clear?

surface, surfaceview, and bufferqueue

Let's break it down with a **clear relationship** between `Surface`, `SurfaceView`, and `BufferQueue` in Android — with **visual explanation** and **simple examples**.

---

## 🧱 1. What is `BufferQueue`?

A **BufferQueue** is like a **pipe** between two ends:

- One side **produces** frames (images)
- One side **consumes** frames (displays them or processes them)

```text
Producer ───> [BufferQueue] ───> Consumer
```

It's the **core graphics mechanism** used in Android.

---

## 🟦 2. What is `Surface`?

A **`Surface`** is a **handle to the producer side** of a **`BufferQueue`**.

- It is what **your app, camera, OpenGL, or video decoder** uses to draw or write frames.
- Internally, **`Surface`** is tightly connected to a **`BufferQueue`**.

🧠 **Every Surface has a BufferQueue internally.**

---

## 🟨 3. What is `SurfaceView`?

A **`SurfaceView`** is a **View** (UI widget) that:

- **Creates its own Surface**
- That Surface is **rendered in a separate window** (outside the regular UI thread)
- Used when you want **fast rendering** (like camera preview, video playback, OpenGL)

```text
SurfaceView
    └── internally creates → Surface
                               └── uses → BufferQueue
```

---

## 🎯 Relation Summary

| Concept | What it is | Related to |
|---|---|---|
| **`BufferQueue`** | Pipe between producer & consumer | Core mechanism |
| **`Surface`** | Producer-side handle to BufferQueue | Wraps a BufferQueue |
| **`SurfaceView`** | View that manages a Surface | Creates a Surface |

---

## 🧠 Example: Showing Camera Preview in SurfaceView

```text
Camera (HAL) → Surface (producer side)
                 → BufferQueue
                     → SurfaceFlinger (consumer side)
                         → Screen
```

- `SurfaceView` creates a `Surface`
- `Camera HAL` gets the producer side of that Surface
- `SurfaceFlinger` is the consumer
- Frames flow from camera to screen via BufferQueue

---

## ✅ Final Notes

- A `SurfaceView` always **creates and manages a Surface**
- A `Surface` always **wraps a BufferQueue**
- A `BufferQueue` always connects **one producer** to **one consumer**

Would you like a diagram based on this explanation?

Graphics architecture

bookmark_border
What every developer should know about surfaces, SurfaceHolder, EGLSurface, SurfaceView, GLSurfaceView, SurfaceTexture, TextureView, SurfaceFlinger, and Vulkan.

This page describes essential elements of the Android system-level graphics architecture and how they are used by the app framework and multimedia system. The focus is on how buffers of graphical data move through the system. If you've ever wondered why SurfaceView and TextureView behave the way they do, or how surfaces and EGLSurface interact, you're in the correct place.

Some familiarity with Android devices and app development is assumed. You don't need detailed knowledge of the app framework and very few API calls are mentioned, but the material doesn't overlap with other public documentation. The goal is to provide details on the significant events involved in rendering a frame for output to help you make informed choices when designing an app. To achieve this, we work from the bottom up, describing how the UI classes work rather than how they can be used.

This section includes several pages covering everything from background material to HAL details to use cases. It starts with an explanation of Android graphics buffers, describes the composition and display mechanism, then proceeds to the higher-level mechanisms that supply the compositor with data. We recommend reading pages in the order listed below rather than skipping to a topic that sounds interesting.

Low-level components
BufferQueue and gralloc. BufferQueue connects something that generates buffers of graphical

data (the producer) to something that accepts the data for display or further processing (the consumer). Buffer allocations are performed through the gralloc memory allocator implemented through a vendor-specific HAL interface.
SurfaceFlinger, Hardware Composer, and virtual displays. SurfaceFlinger accepts buffers of data from multiple sources, composites them, and sends them to the display. The Hardware Composer HAL (HWC) determines the most efficient way to composite buffers with the available hardware, and virtual displays make composited output available within the system (recording the screen or sending the screen over a network).
Surface, canvas, and SurfaceHolder. A surface produces a buffer queue that is often consumed by SurfaceFlinger. When rendering onto a surface, the result ends up in a buffer that gets shipped to the consumer. Canvas APIs provide a software implementation (with hardware-acceleration support) for drawing directly on a surface (low-level alternative to OpenGL ES). Anything having to do with a view involves a SurfaceHolder, whose APIs enable getting and setting surface parameters such as size and format.
EGLSurface and OpenGL ES. OpenGL ES (GLES) defines a graphics-rendering API designed to be combined with EGL, a library that can create and access windows through the operating system (to draw textured polygons, use GLES calls; to put rendering on the screen, use EGL calls). This page also covers ANativeWindow, the C/C++ equivalent of the Java Surface class used to create an EGL window surface from native code.
Vulkan. Vulkan is a low-overhead, cross-platform API for high-performance 3D graphics. Like OpenGL ES, Vulkan provides tools for creating high-quality, real-time graphics in apps. Vulkan advantages include reductions in CPU overhead and support for the SPIR-V Binary Intermediate language.
High-level components
SurfaceView and GLSurfaceView. SurfaceView combines a surface and a view. SurfaceView's view components are composited by SurfaceFlinger (and not the app), enabling rendering from a separate thread/process and isolation from app UI rendering. GLSurfaceView provides helper classes to manage EGL contexts, interthread communication, and interaction with the activity lifecycle (but isn't required to use GLES).
SurfaceTexture. SurfaceTexture combines a surface and GLES texture to create a BufferQueue for which your app is the consumer. When a producer queues a new buffer, it notifies your app, which in turn releases the previously-held buffer, acquires the new buffer from the queue, and makes EGL calls to make the buffer available to GLES as an external texture. Android 7.0 added support for secure texture video playback enabling GPU post-processing of protected video content.
TextureView. TextureView combines a view with a SurfaceTexture. TextureView wraps a SurfaceTexture and takes responsibility for responding to callbacks and acquiring new buffers. When drawing, TextureView uses the contents of the most recently received buffer as its data source, rendering wherever and however the view state indicates it should. View composition is always performed with GLES, meaning updates to contents may cause other view elements to redraw as well.

Great — this summary captures key components of Android’s graphics architecture. Let me break down **Surface**, **SurfaceView**, **BufferQueue**, and how they relate with each other in **simple, layered explanation**, and tie them into the bigger picture described above.

# 🔹 Core Concepts Overview

## 🧱 1. BufferQueue – the backbone

- Think of **BufferQueue** as a **pipe with a queue of graphic buffers**.
- It connects a **producer** (who draws or fills buffers) to a **consumer** (who displays or processes them).
- The producer could be:
  - An app rendering via OpenGL ES, Vulkan, or Canvas.
  - The camera HAL providing camera frames.
- The consumer could be:
  - SurfaceFlinger (for screen display)
  - GPU (for texture)
  - MediaCodec, Camera HAL, etc.

### ⚙️ Contains:

- Graphic buffers (gralloc handles)
- Queue management logic
- Sync primitives (fences, semaphores)

---

## 📱 2. Surface

- A **Surface is just a high-level abstraction for the producer side of a BufferQueue**.
- When you draw to a Surface (Canvas, GLES, Vulkan), you're submitting buffers to its BufferQueue.

**Surface holds:**

- Reference to the **producer** side of a BufferQueue
- Used by apps to render graphics
- Native equivalent: `ANativeWindow`

---

## 3. SurfaceView

- A **special View that owns a Surface**.
- Unlike normal Views, which are drawn into the View hierarchy’s buffer, **SurfaceView gets its own separate Surface** (meaning its own **BufferQueue**).

- This **Surface is composited separately** by **SurfaceFlinger**, not by the app's UI thread.

**Benefits:**

- Ideal for rendering high-frame-rate or external content (e.g., video, camera)
- Reduces UI thread overhead
- Hardware-accelerated rendering with GLES/Vulkan

---

### 🌀 4. TextureView

- Like SurfaceView but lives **inside** the regular view hierarchy.
- Internally uses a `SurfaceTexture` (which is both a producer and consumer) that allows the view system to access GPU textures.
- You can manipulate and transform it like any other View (scaling, rotation, etc.)

---

## 🧩 How They Work Together

### ➤ App Rendering Path (Simple):

```text
App → Surface (produces buffer) → BufferQueue → SurfaceFlinger (consumes buffer) → Display
```

### ➤ SurfaceView Path:

```text
App creates SurfaceView
↳ SurfaceView gets its own Surface
↳ That Surface has its own BufferQueue
↳ Your thread draws to the Surface
↳ Buffers go to SurfaceFlinger
↳ Composited onto display
```

### ➤ TextureView Path:

```text
App creates TextureView
↳ Internally uses SurfaceTexture
↳ SurfaceTexture gets updates via GLES texture
↳ TextureView updates texture and draws into View hierarchy
```

---

# 📌 How Many BufferQueues Are Created?

Each Surface or SurfaceTexture has **its own BufferQueue**.

- `SurfaceView` → **1 BufferQueue**
- `GLSurfaceView` → **1 BufferQueue**
- `MediaCodec` output surface → **1 BufferQueue**
- `Camera HAL` preview stream → **1 BufferQueue**

Each **pair of producer-consumer** requires its **own BufferQueue**.

---

# 🔁 Quick Summary Comparison

| Component | Own Surface? | View Hierarchy? | BufferQueue? | Composited By |
|---|---|---|---|---|
| `Surface` | Yes | No | Yes | Depends (e.g. SF) |
| `SurfaceView` | Yes | Partially | Yes | SurfaceFlinger |
| `TextureView` | No (uses ST) | Yes | Yes (ST) | View System (GLES) |
| `GLSurfaceView` | Yes | No | Yes | SurfaceFlinger |
| `SurfaceTexture` | Yes | No | Yes | GLES |

---

Would you like a visual diagram to make this even clearer?

relation of grafficbuffer with bufferqueue

Let's break this down **simply and clearly**:

---

# 🔗 Relationship between `GraphicBuffer` and `BufferQueue`

## 🧱 1. What is `GraphicBuffer`?

- `GraphicBuffer` is a **data structure** that wraps a block of memory used to hold **pixel data** (e.g., an image frame).
- Think of it as a **frame-sized container** where graphics or video content is drawn or decoded.
- Internally, it manages a **GPU-allocated buffer** (usually via `gralloc`).

👉 It's used across the Android graphics stack for sharing memory between processes (like between an app and SurfaceFlinger).

---

### 🔲 2. What is `BufferQueue`?

- **`BufferQueue`** is a **communication pipe** between a producer (e.g. app or camera) and a consumer (e.g. SurfaceFlinger or GPU).
- It **holds multiple GraphicBuffers** in a queue so that:
  - The producer can write to one buffer
  - The consumer can read from another
- It handles **synchronization**, **buffer reuse**, and **lifetime management**.

---

## 🔄 How are they related?

- **`BufferQueue`** contains a **set of `GraphicBuffer` objects** internally.
- When a producer (e.g., Surface, SurfaceView, SurfaceTexture) needs to render:
  - It **dequeues** a **`GraphicBuffer`** from the **`BufferQueue`**
  - It **draws into it**
  - Then **queues it back** into the **`BufferQueue`**
- The consumer (like SurfaceFlinger or a TextureView) then **acquires** this **`GraphicBuffer`** and displays or processes it.

---

### 📦 Visualization:

```plaintext
App (producer)         BufferQueue                SurfaceFlinger (consumer)
  ┌──────────┐        ┌────────────┐           ┌───────────────────┐
  │ draw()   │  ──►   │ GraphicBuf1│ ────────► │ display on screen │
  │          │  ◄──   │ GraphicBuf2│ ◄──────── │                   │
  └──────────┘        └────────────┘           └───────────────────┘
```

---

### 📘 Summary:

| Concept | Role |
|---|---|
| **`GraphicBuffer`** | Memory block for pixel/frame data |
| **`BufferQueue`** | Manages and transfers a queue of GraphicBuffers |
| Relation | **`BufferQueue`** manages multiple **`GraphicBuffers`** |

Let me know if you'd like to go deeper into **`gralloc`**, **`Fence`** sync, or where SurfaceView and TextureView fit in this flow.

> why surface is at producer end of bufferqueue

Great question — and **yes, it can be confusing** at first!
Let's break it down simply:

---

## ❓ Why is `Surface` the Producer in a `BufferQueue`?

### 🔁 Quick Reminder:

- A **`BufferQueue`** connects:
  - **Producer** → who **draws/writes** pixels (e.g., app rendering)
  - **Consumer** → who **reads/displays** pixels (e.g., SurfaceFlinger)

---

## 📱 What is `Surface`?

- **`Surface`** is an **interface** that apps use to draw or render graphics.
- It can be backed by:
  - A **`SurfaceView`**
  - A **`TextureView`**
  - Or used directly in native code (e.g., OpenGL, Vulkan, Camera)

Internally, the **`Surface`** holds the **producer side** of a **`BufferQueue`**.

---

## 🎯 Why is `Surface` the producer?

Because **drawing into a Surface** means you are **producing image frames**.

### ✅ Your app does this:

```java
Surface surface = surfaceView.getHolder().getSurface();
Canvas canvas = surface.lockCanvas();
```

```
// Draw something...
surface.unlockCanvasAndPost(canvas);
```

🔄 This is:

- `lockCanvas()` → **dequeue a GraphicBuffer from BufferQueue**
- Draw → **write pixels**
- `unlockCanvasAndPost()` → **queue it back to BufferQueue**

Thus:
👉 **`Surface` is writing image data = producer**
👉 The **consumer** (like SurfaceFlinger) will read this and **display it**

---

## 📦 What's on the Consumer Side?

- Usually: **`SurfaceFlinger`**
  - It reads the buffer, composes the final screen frame, and sends it to the display
- Sometimes: **`GLConsumer`**, **`MediaCodec`**, etc. (for video, camera, etc.)

---

## 🧠 Analogy:

> Think of **`Surface`** like a **paintbrush**:
> You dip the brush into a canvas (GraphicBuffer), paint (render), then send it to a gallery (SurfaceFlinger) to hang on the wall (display).

---

## ✅ Summary

| Role | Component | What it does |
|---|---|---|
| **Producer** | **`Surface`** | App draws into it (produces frames) |
| **BufferQueue** | Shared queue | Holds buffers during transfer |
| **Consumer** | **`SurfaceFlinger`**, **`GLConsumer`** | Displays or uses the frames |

---

Let me know if you want to visualize how **`SurfaceView`**, **`SurfaceTexture`**, or **`Camera`** fit into this!

BufferQueue and Gralloc

bookmark_border

The BufferQueue class connects components that generate buffers of graphical data (producers) to components that accept the data for display or further processing (consumers). Nearly everything that moves buffers of graphical data through the system relies on BufferQueue.

The Gralloc memory allocator performs buffer allocations and is implemented through two vendor-specific HIDL interfaces (see hardware/interfaces/graphics/allocator/ and hardware/interfaces/graphics/mapper/). The allocate() function takes expected arguments (width, height, pixel format) as well as a set of usage flags.

BufferQueue producers and consumers
Consumers create and own the BufferQueue data structure and can exist in different processes than their producers. When a producer needs a buffer, it requests a free buffer from BufferQueue by calling dequeueBuffer(), specifying the buffers's width, height, pixel format, and usage flags. The producer then populates the buffer and returns the buffer to the queue by calling queueBuffer(). Next, the consumer acquires the buffer with acquireBuffer() and makes use of the buffer contents. When the consumer is done, it returns the buffer to the queue by calling releaseBuffer(). The sync framework controls how buffers move through the Android graphics pipeline.

Some characteristics of the BufferQueue, such as the maximum number of buffers it can hold, are determined jointly by the producer and the consumer. However, BufferQueue allocates buffers as it needs them. Buffers are retained unless the characteristics change; for example, if a producer requests buffers with a different size, old buffers are freed and new buffers are allocated on demand.

Buffer contents are never copied by BufferQueue, as moving that much data around is inefficient. Instead, buffers are always passed by a handle.

Track BufferQueue with Systrace
To understand how graphics buffers move around, use Systrace, a tool that records device activity over a short period of time. The system-level graphics code is well instrumented, as is much of the relevant app framework code.

To use Systrace, enable the gfx, view, and sched tags. BufferQueue objects are displayed in the trace. As an example, if you take a trace while Grafika's Play video (SurfaceView) is running, the row labeled SurfaceView tells you how many buffers were queued up at any given time.

The value increments while the app is active, which triggers the rendering of frames by the MediaCodec decoder. The value decrements while SurfaceFlinger is working and consuming buffers. When showing video at 30 fps, the queue's value varies from 0 to 1 because the ~60 fps display can keep up with the source. SurfaceFlinger wakes only when there's work to be done, not 60 times per second. The system tries to avoid work and disables VSync if nothing is updating the screen.

If you switch to Grafika's Play video (TextureView) and grab a new trace, you see a row labeled com.android.grafika/com.android.grafika.PlayMovieActivity. This is the main UI layer, which is another BufferQueue instance. Because TextureView renders into the UI layer rather than a separate layer, all of the video-driven updates are displayed here.

Gralloc
The Gralloc allocator HAL hardware/libhardware/include/hardware/gralloc.h performs buffer allocations through usage flags. Usage flags include attributes such as:

How often the memory will be accessed from software (CPU)
How often the memory will be accessed from hardware (GPU)
Whether the memory will be used as an OpenGL ES (GLES) texture
Whether the memory will be used by a video encoder
For example, if a producer's buffer format specifies RGBA_8888 pixels, and the producer indicates that the buffer will be accessed from software (meaning an app will touch pixels on the CPU), Gralloc creates a buffer with 4 bytes per pixel in R-G-B-A order. If instead, a producer specifies its buffer will be only accessed from hardware and as a GLES texture, Gralloc can do anything the GLES driver wants, such as BGRA ordering, nonlinear swizzled layouts, and alternative color formats. Allowing the hardware to use its preferred format can improve performance.

Some values can't be combined on certain platforms. For example, the video encoder flag may require YUV pixels, so adding software access and specifying RGBA_8888 fails.

The handle returned by Gralloc can be passed between processes through Binder.

Protected buffers
The Gralloc usage flag GRALLOC_USAGE_PROTECTED allows the graphics buffer to be displayed only through a hardware-protected path. These overlay planes are the only way to display DRM content (DRM-protected buffers can't be accessed by SurfaceFlinger or the OpenGL ES driver).

DRM-protected video can be presented only on an overlay plane. Video players that support protected content must be implemented with SurfaceView. Software running on unprotected hardware can't read or write the buffer; hardware-protected paths must appear on the Hardware Composer overlay (that is, protected videos disappear from the display if Hardware Composer switches to OpenGL ES composition).

For details on protected content, see DRM.

Let's break down this content into **simple concepts** with a **clear flow** so it becomes less confusing.

---

## 🧱 Key Components

### 1. BufferQueue

- Think of **BufferQueue** as a **mailbox** used to **pass image frames** (buffers) between two parts of the system:
  - A **Producer** (like the camera or video decoder)
  - A **Consumer** (like SurfaceFlinger or a video display window)
- 🔄 **Flow of a buffer**:

```arduino
Producer           BufferQueue              Consumer
   │                   │                       │
   ├──dequeueBuffer()→ │                       │  ← Ask for empty buffer
   ├──write pixels     │                       │
   ├──queueBuffer()→   │                       │  ← Put buffer into queue
   │                   ├──acquireBuffer()→     │  ← Get buffer to display
   │                   ├──display/use it       │
   │                   ├──releaseBuffer()→     │  ← Return buffer back
```

- 📦 Buffers are **not copied** — only **passed by handle**, making this fast.

---

## 2. Gralloc (Graphics Allocator)

- Gralloc is a **low-level memory manager** that **allocates buffers** for images using special hardware-friendly memory.
- It is used **by BufferQueue** when it needs to allocate buffers.
- 📋 Gralloc decides:
  - **Where** memory will be used: CPU or GPU
  - **How** memory is formatted: RGBA, YUV, etc.
  - **What** it will be used for: OpenGL texture, video encoder, etc.
- 🧠 It's like telling a chef:

  > "Give me a tray that's 1920x1080 pixels, in RGBA format, and I'll use it with GPU for video rendering."

---

# 🧵 BufferQueue + Gralloc Workflow

1. **Consumer** creates the BufferQueue.
2. **Producer** asks for a buffer from BufferQueue → **`dequeueBuffer()`**.
3. If no suitable buffer exists:
   - **BufferQueue uses Gralloc to allocate** one, based on:
     - size

- format (RGBA/YUV)
- usage (CPU/GPU access, video encoder, etc.)

4. Producer fills the buffer (e.g., camera writes pixels).
5. Producer calls **`queueBuffer()`** → passes it to the consumer.
6. Consumer calls **`acquireBuffer()`** → gets the buffer.
7. Consumer displays/processes it and calls **`releaseBuffer()`**.

---

# 🎥 Example: Video Playback

### Case 1: SurfaceView

- Video decoded by **MediaCodec** into buffers.
- Buffers are handed to SurfaceFlinger **via BufferQueue**.
- If content is **DRM-protected**, it **must** go through **hardware overlay** using:

```nginx
GRALLOC_USAGE_PROTECTED
```

### Case 2: TextureView

- Video drawn to a UI texture, so it's part of the app's main UI layer.
- Frame updates are seen in the app's **UI BufferQueue**.

---

# 🔐 Protected Buffers

- **Why?** To prevent apps from **reading or copying** DRM content (Netflix, Prime Video).
- **How?**
  - Gralloc uses a **"protected" flag**.
  - Only special **hardware overlays** can display these.
  - OpenGL and SurfaceFlinger **cannot touch** them.

---

# 📊 Debugging: Systrace

- You can **trace how buffers move** using **Systrace** tool:
  - Enable **`gfx`**, **`view`**, and **`sched`** tags.

- Shows each **`queueBuffer`**, **`acquireBuffer`**, etc.
- Visualizes **when** producer/consumer are working.

---

## 🔁 Summary in 3 Lines

| Component | Role | Who Uses It? |
|---|---|---|
| BufferQueue | Connects producer → consumer | Camera, SurfaceFlinger, UI, VideoView |
| Gralloc | Allocates GPU/CPU-friendly memory | Used by BufferQueue internally |
| Protected Buffers | Secure rendering for DRM content | Requires hardware overlays |

---

If you want a **visual flow diagram** or example trace log explained, let me know!

Implement DMABUF and GPU memory accounting in Android 12

bookmark_border
This page describes the various memory accounting improvements introduced in Android 12.

DMA-BUF statistics in sysfs
In Android 11 and Android 12, debugfs can't be mounted in User builds. So DMA-BUF statistics has been added to sysfs in the /sys/kernel/dmabuf/buffers directory in Android 12.

Path Description
/sys/kernel/dmabuf/buffers The /sys/kernel/dmabuf/buffers directory contains a snapshot of the internal state of every DMA-BUF. /sys/kernel/dmabuf/buffers/<inode_number> contains the statistics for the DMA-BUF with the unique inode number <inode_number>.
/sys/kernel/dmabuf/buffers/<inode_number>/exporter_name This read-only file contains the name of the DMA-BUF exporter.
/sys/kernel/dmabuf/buffers/<inode_number>/size This read-only file specifies the size of the DMA-BUF in bytes.
The libdmabufinfo API parses the DMA-BUF sysfs stats to expose per-exporter and per-buffer statistics.

Please note that kernel drivers who export DMA-BUFs must set the exp_name field of struct dma_buf_export_info correctly to the exporter name before invoking the dma_buf_export() API to create a DMA-BUF. This is required for libdmabufinfo and the dmabuf_dump tool to derive per-exporter statistics which are then exposed in bugreport.

The dmabuf_dump tool has been modified to output this information with a new argument, -b.

Statistics for the DMA-BUF heaps framework

ION in GKI 2.0 is being deprecated in favor of the DMA-BUF heaps framework, which is part of the upstream linux kernel.

The following global ION statistics are tracked in Android 11:

Total size of DMA-BUFs exported by every ION heap
Total size of unused pre-allocated memory stored by every ION heap
There is no interface available to expose per-ION heap statistics in Android 11.

The following table compares the ION statistics interfaces with their counterparts for devices that use the DMA-BUF heap framework in Android 12.

Android 11 or Devices launching with ION support in Android 12 Devices launching with DMA-BUF heaps in Android 12
Per-heap ION statistics None Parsed from DMA-BUF sysfs stats
Total size of DMA-BUFs exported /sys/kernel/ion/total_heap_size_kb
(Doesn't include the size of DMA-BUFs exported by non-ION exporters) Parsed from DMA-BUF sysfs stats
(includes the size of all DMA-BUFs exported).
Total memory pooled by heaps /sys/kernel/ion/total_pool_size_kb
/sys/kernel/dma_heap/total_pool_size_kb
Improve the lost RAM calculation accuracy
Previously the lost RAM calculation was done as follows:

final long lostRAM = memInfo.getTotalSizeKb() - (totalPss - totalSwapPss)

- memInfo.getFreeSizeKb() - memInfo.getCachedSizeKb()

- kernelUsed - memInfo.getZramTotalSizeKb();

The totalPss component included the GPU memory usage (returned by the Memtrack HAL's getMemory() interface). The kernelUsed component included the total DMA-BUF memory usage. However, for Android devices, the GPU memory came from the following:

Direct allocations made by the GPU driver using physical page allocator
DMA-BUFs mapped into GPU address space
Therefore, DMA-BUFs that were memory-mapped into the GPU address space were subtracted twice when lost RAM was calculated. Android 12 implements a solution to calculate the size of DMA-BUFs mapped into the GPU address space, which means that it's accounted for only once in the Lost RAM calculation.

The details of the solution are as follows:

The Memtrack HAL API getMemory() when called with PID 0 must report the global total GPU-private memory, for MemtrackType::GL and MemtrackRecord::FLAG_SMAPS_UNACCOUNTED.
getMemory() when called with PID 0 for a MemtrackType other than GL must not fail. It must

instead return 0.
The GPU memory tracepoint/eBPF solution added in Android 12 accounts for total GPU memory. Subtracting the total GPU private memory from the total GPU memory provides the size of DMA-BUFs mapped into the GPU address space. The value can then be used to improve the accuracy of Lost RAM calculations by correctly accounting for the GPU memory usage.
The private GPU memory is included in totalPss in most Memtrack HAL implementations and therefore must be deduplicated before removing it from lostRAM.
The implemented solution is detailed in the next section.

Remove Memtrack variability from lost RAM
Since Memtrack HAL implementations can vary across partners, the GPU memory included in totalPSS from the HAL isn't always consistent. To remove the variability from lostRAM, the memory accounted for in MemtrackType::GRAPHICS and MemtrackType::GL is removed from totalPss during the lostRAM calculation.

MemtrackType::GRAPHICS memory is removed from totalPss and replaced with the totalExportedDmabuf memory in the lostRAM calculation in ActivityManagerService.java as shown below:

```
final long totalExportedDmabuf = Debug.getDmabufTotalExportedKb();

. . .

final long dmabufUnmapped = totalExportedDmabuf - dmabufMapped;

. . .

// Account unmapped dmabufs as part of the kernel memory allocations
kernelUsed += dmabufUnmapped;

// Replace Memtrack HAL reported Graphics category with mapped dmabufs
totalPss -= totalMemtrackGraphics;
totalPss += dmabufMapped;
```

MemtrackType::GL memory is removed from totalPss and replaced with the private GPU memory (gpuPrivateUsage) in lostRAM calculation in ActivityManagerService.java as shown below:

```
final long gpuUsage = Debug.getGpuTotalUsageKb();

. . .

final long gpuPrivateUsage = Debug.getGpuPrivateMemoryKb();

. . .
```

// Replace the Memtrack HAL-reported GL category with private GPU allocations.
// Count it as part of the kernel memory allocations.
totalPss -= totalMemtrackGl;
kernelUsed += gpuPrivateUsage;
Updated lost RAM calculation
Both the total private GPU memory and the total exported DMA buffer memory are contained in kernelUsed + totalPss which is removed from lostRAM. This eliminates both double-counting and Memtrack variability from lost RAM calculation.

final long lostRAM = memInfo.getTotalSizeKb() - (totalPss - totalSwapPss)
- memInfo.getFreeSizeKb() - memInfo.getCachedSizeKb()
- kernelUsed - memInfo.getZramTotalSizeKb();
Validation
VTS tests enforce the rule that devices launching in Android 12 with a Linux kernel version 5.4 or higher support the getGpuDeviceInfo() API.

A new Memtrack HAL API getGpuDeviceInfo() must return information about the GPU device in use.

This provides better memory accounting and visibility into DMA buffer and GPU memory usage. Implement the memtrack AIDL HAL for better lost RAM and memory accounting. This feature isn't dependent on Google services.

Implementation
This feature depends on the AIDL Memtrack HAL, and directions for implementing it in Android 12 are included in the code as comments.

All HIDL HALs are planned to be converted to AIDL in future releases.

The following APIs have been added to core/java/android/os/Debug.java:

```
 /**
  * Return total memory size in kilobytes for exported DMA-BUFs or -1 if
  * the DMA-BUF sysfs stats at /sys/kernel/dmabuf/buffers could not be read.
  *
  * @hide
  */
 public static native long getDmabufTotalExportedKb();

 /**
  * Return memory size in kilobytes allocated for DMA-BUF heap pools or -1 if
  * /sys/kernel/dma_heap/total_pools_kb could not be read.
  *
```

```
 * @hide
 */
public static native long getDmabufHeapPoolsSizeKb();
```

To ensure your version works as intended, integrate the tracepoints in your GPU drivers, and implement the AIDL memtrack HAL getMemory() API to correctly return the global total GPU-private memory when called with PID 0 for MemtrackType::GL and MemtrackRecord::FLAG_SMAPS_UNACCOUNTED.

Here's a **summary and breakdown** of how **DMA-BUF and GPU memory accounting** is improved in **Android 12**, and how components like `BufferQueue`, `Gralloc`, and the kernel work together for **accurate RAM usage tracking** — especially to avoid *double-counting* and *inconsistent memory reports*.

---

## 🔍 1. Why These Changes?

**Problem in earlier versions:**

- GPU memory used via DMA-BUFs was **counted twice**:
  - Once as part of **Memtrack HAL GPU reports**
  - Once as part of **kernel DMA-BUF memory**
- **DebugFS** was inaccessible on user builds (Android 11), limiting visibility.
- **ION allocator** lacked detailed per-heap stats.

---

## 🧠 2. What's New in Android 12?

### ✅ DMA-BUF stats exposed via `sysfs` (instead of `debugfs`)

- Located at: `/sys/kernel/dmabuf/buffers/`
- For each DMA-BUF:
  - `/exporter_name` → Who exported it (Gralloc, GPU driver, etc.)
  - `/size` → Buffer size in bytes
- Enables **libdmabufinfo** and **dmabuf_dump -b** to show per-exporter stats
- Exporters must set `exp_name` in `dma_buf_export_info`

---

### ✅ Switch from ION to DMA-BUF heaps

| Feature | Android 11 (ION) | Android 12 (DMA-BUF Heaps) |
|---|---|---|
| Per-heap stats | ❌ None | ✅ Parsed from **`/sys/kernel/dmabuf/buffers`** |
| Total exported size | **`/sys/kernel/ion/total_heap_size_kb`** | ✅ Derived from DMA-BUF sysfs |
| Total pooled memory | **`/sys/kernel/ion/total_pool_size_kb`** | **`/sys/kernel/dma_heap/total_pool_size_kb`** |

---

## ✅ Accurate "Lost RAM" Calculation

Fixes the problem of **double-counting DMA-BUFs** mapped into GPU memory.

New logic:

- Use **`Memtrack HAL (AIDL)`** to report:
  - **`GPU-private memory`**: via **`getMemory(PID=0, MemtrackType::GL)`**
- Use tracepoints/eBPF to get:
  - **`Total GPU memory usage`**
- **Subtract private GPU memory** from total GPU usage → Gives **`DMA-BUF mapped size`**
- Use this in RAM calculation.

## 🔄 Old vs New Lost RAM Equation

### Old (buggy):

```java
lostRAM = totalSizeKb - (totalPss - swapPss) - freeKb - cachedKb - kernelUsed - zramSize
```

### New (accurate):

```java
// Swap out Memtrack graphics/gl stats with accurate values
totalPss -= totalMemtrackGraphics;
totalPss += dmabufMapped;

totalPss -= totalMemtrackGL;
kernelUsed += gpuPrivateUsage;

lostRAM = totalSizeKb - (totalPss - swapPss) - freeKb - cachedKb - kernelUsed - zramSize;
```

---

# 🔌 3. APIs and Tools

## 🔧 Android APIs Added (`android.os.Debug`)

```java
public static native long getDmabufTotalExportedKb();
public static native long getDmabufHeapPoolsSizeKb();
```

## 🧰 Tool: `dmabuf_dump -b`

- Prints buffer sizes by exporter

## 🔍 Systrace Tags

- `gfx`, `view`, `sched` — to visualize buffer movement

---

# ✅ 4. Validation Requirements for Android 12 Devices

| Requirement | Applicable Devices |
|---|---|
| AIDL Memtrack HAL | Android 12+ with kernel ≥ 5.4 |
| `getMemory(PID=0, MemtrackType::GL)` must work | ✅ Required |
| `getGpuDeviceInfo()` must be implemented | ✅ Required |
| GPU driver must support tracepoints | ✅ Required |

---

# 🔄 5. Summary: Component Roles

| Component | Role in DMA-BUF / Memory Tracking |
|---|---|
| `BufferQueue` | Handles buffer exchange between producers & consumers |
| `Gralloc` | Allocates memory for DMA-BUFs |
| `DMA-BUF` | Shareable memory handle for buffers across subsystems |
| `Memtrack HAL` | Reports GPU memory usage, now split between private and DMA-BUF |
| `sysfs (/sys/kernel/dmabuf/buffers)` | Exposes per-buffer memory stats |
| `eBPF / Tracepoints` | Tracks actual GPU memory usage |

---

If you'd like:

- A **diagram** of the new memory accounting pipeline
- A **sample trace walkthrough**

- Example output from `dmabuf_dump` or Memtrack HAL

Let me know and I can generate it for you.

dma-buf vs ion

### DMA-BUF vs ION in Android/Linux Memory Sharing

Both **DMA-BUF** and **ION** are **kernel-level mechanisms** used in Android/Linux to **share memory buffers between different subsystems** like GPU, camera, video codecs, and display. However, they are **not the same**, and their roles have evolved over time.

---

## 🧠 TL;DR

| Feature | ION | DMA-BUF |
|---|---|---|
| Purpose | Android-specific buffer allocator | Standard Linux kernel buffer sharing |
| Buffer Sharing | Indirect (uses custom interfaces) | Direct (via file descriptors) |
| Security | Weaker isolation, custom | Stronger isolation, unified model |
| Lifetime Mgmt | Custom handle-based | File descriptor-based (reference counted) |
| Deprecated? | Partially deprecated in AOSP | Actively used, preferred in new systems |
| Used for | Android HALs, older devices | All modern Android devices |

---

### ✅ What is ION?

- ION (Android **I**nput-**O**utput **N** memory allocator) was introduced to support memory sharing between Android multimedia components (camera, video, GPU).
- It provides:
  - **Memory allocation**
  - **Heap types** (like system, secure, DMA)
  - **Buffer sharing via custom ION handles**
- It abstracts memory allocation from clients. Vendors would implement custom heaps for specific hardware.

🔸 **Problems with ION:**

- Android-specific, not upstreamed in Linux.
- Multiple incompatible ION heap implementations.

- Poor security (shared handles could be misused).
- No standard buffer lifetime tracking (manual).

---

## ✅ What is DMA-BUF?

- DMA-BUF is a **Linux kernel standard mechanism** for **zero-copy buffer sharing** between devices (GPU, camera, display, etc).
- Works via **file descriptors**: a DMA-BUF is a file descriptor that refers to a buffer.
- Used with **`EXPORT`** and **`IMPORT`** model between devices:
  - One driver **exports** a buffer
  - Another driver **imports** and accesses the same physical memory

### 🔒 Advantages of DMA-BUF:

- File-descriptor based = better reference counting, memory lifetime management
- Kernel standard = upstreamed, portable, supported widely (Linux, Android, DRM/KMS, etc)
- Works well with secure memory (like DRM-protected video)
- Avoids redundant copies by allowing subsystems to operate on same memory

---

## 🔁 ION vs DMA-BUF Usage

- In modern Android (Android 8+), **DMA-BUF is the preferred way** to share buffers.
- ION is **still used** for **allocation**, but often the allocated buffers are **exported as DMA-BUF fds**.
- Example: Camera HAL may allocate a buffer using ION, but pass a DMA-BUF fd to the GPU or display.
- The Android **Gralloc HAL** typically handles this transition.

---

## 🔧 How They're Connected in Android

1. **Allocation:**
   - **`ION_IOC_ALLOC`** → Allocate memory (legacy method).
   - Gralloc HAL may use **`ion_alloc_fd`** to get a DMA-BUF file descriptor.
2. **Usage:**
   - GPU or display receives **fd (DMA-BUF)** via **`ANativeWindow`** / **`BufferQueue`**.
3. **Sharing:**

- Different devices (e.g., GPU + Display + Codec) map/import the **same DMA-BUF** into their memory space.

---

## ✅ Summary

- **ION**: Android-specific allocator, legacy, vendor-customizable
- **DMA-BUF**: Linux-standard, secure, zero-copy memory sharing
- **Modern Android**: DMA-BUF preferred for sharing, ION may still be used for allocation

---

If you want, I can show you how a DMA-BUF is passed from one driver to another using file descriptors, or how Gralloc uses both.

relation of heap wiht these. explain in simple words

Great question!

Let's explain the **relation of "heaps"** with **ION** and **DMA-BUF**, in **simple words**:

---

### 🔹 What is a Heap (in this context)?

In this case, **"heap"** means a **type of memory pool** — a source of memory from which buffers are allocated.

Just like:

> 💡 "You take water from different tanks (cold, hot, clean, dirty) depending on your need"

→ Similarly, in Android/Linux, **you take memory from different heaps depending on what you're doing** (GPU, camera, display, etc).

---

### 🔹 In ION, what is a heap?

In **ION**, a **heap** is a specific **type of memory region** that ION can allocate from.

For example:

| Heap Type | Used For |
|---|---|
| **`system`** heap | Generic memory (fallback) |
| **`system-contig`** | Physically contiguous memory |
| **`dma`** heap | For DMA hardware access |

| Heading Type | Used For |
|---|---|
| **`secure`** heap | For protected video memory (DRM) |
| **`custom`** heaps | Vendor-specific (camera, GPU, etc) |

So when you call ION to allocate a buffer, you must **choose a heap**:

```c
ion_alloc(ion_fd, size, heap_type, flags);
```

You tell it:
→ "Give me a buffer from the **camera heap**"
→ or "Give me memory from the **secure heap**"

---

## 🔹 In DMA-BUF, are there heaps?

DMA-BUF **does not manage heaps**.

Instead:

- It just **shares existing buffers** (allocated by something else, like ION or DRM).
- The memory might **originally come from a heap**, but DMA-BUF doesn't care where—it just **wraps the memory into a file descriptor (`fd`)**.

Think of DMA-BUF as a **"container to share"** an already allocated buffer.

---

## 🔄 Summary: ION, DMA-BUF, and Heap

| Concept | Role |
|---|---|
| **Heap** | A memory pool used to allocate buffers |
| **ION** | Allocates from heaps (system, secure, etc) |
| **DMA-BUF** | Shares buffers (regardless of heap) |

➡️ Typically in Android:

1. App → Camera → ION allocates buffer from **`camera heap`**
2. That buffer is turned into a **DMA-BUF file descriptor**
3. That **`fd`** is passed to GPU or Display via **BufferQueue**
4. Other drivers use that **shared buffer** without copying

---

Let me know if you want a diagram or a real-world analogy (like parcel delivery).

issue with ion. explain in simple words

Sure! Let's explain the **issues with ION** in **simple words**, and why Android moved away from it.

---

### 🔸 What is ION again?

**ION** was a memory allocator used in Android to:

- Allocate buffers from **different memory heaps**
- **Share** those buffers between camera, GPU, display, etc.

But it had **problems**. Here's what went wrong:

---

## 🔴 1. Too many custom ION heaps

> 🧱 Problem: **Vendors kept adding their own heap types**, like **`qcom_camera_heap`**, **`mediatek_disp_heap`**, etc.

- No standard rules
- Each device had **different heap IDs**
- Hard to write **portable code** (code that works across all devices)

---

## 🔴 2. Security issues

> 🔓 Problem: **Improper buffer access control**

- A buffer allocated by one component (say, camera) could be **accessed by others** (like apps or GPU) **without proper permission checks**.
- Example: A **secure video buffer** might accidentally be readable by untrusted code — that's a **big security hole**.

---

## 🔴 3. Hard to maintain

> 🧩 Problem: ION had a **complex kernel driver**, and every vendor changed it.

- Different OEMs changed ION kernel code
- This made **mainline kernel integration impossible**
- Android kernel updates became harder

---

# 🔴 4. Not compatible with upstream Linux

> 🧱 Problem: ION was **Android-specific**

- The **mainline Linux kernel community** didn't accept ION
- So Android and Linux **diverged** in how memory sharing was done

---

### ✅ What replaced ION?

> **DMA-Heap** is the replacement.

- Cleaner design
- Fewer types of heaps (like `system`, `secure`)
- Follows Linux upstream standards
- Still works with **DMA-BUF** for buffer sharing

Now most Android devices use **DMA-Heap** + **DMA-BUF**, not ION.

---

### 🔄 In Summary:

| Issue | Why it was bad |
| --- | --- |
| Too many custom heaps | Broke compatibility across devices |
| Security holes | Buffers could be misused |
| Not upstreamed | Android diverged from Linux |
| Maintenance burden | OEMs had to maintain custom hacks |

---

Let me know if you'd like to see how ION was used in code or how DMA-Heap fixes it.

DMA-Heap vs DMA-BUF

Great question! Let's break down the difference between **DMA-Heap** and **DMA-BUF** in **very simple words**, and how they work **together**.

---

# 🧱 First, what's the difference?

| Feature | DMA-Heap 🏗️ | DMA-BUF 📦 |
| --- | --- | --- |
| What it does | Allocates memory buffers | Shares those buffers between devices |
| Role | **Creates** the memory | **Passes** memory between components |
| Used by | Camera, GPU, video decoders | Display, GPU, camera, etc. |
| Lifecycle | Allocates memory from a heap | Exports that memory as a shareable handle |

---

### 🎯 Think of it like this:

> 💡 **DMA-Heap = Factory**
> It **creates** the buffer (like a box)
>
> 💡 **DMA-BUF = Shipping label**
> It **shares** the box with other devices like GPU or display

---

## ✅ Real-world analogy:

- 🧱 **DMA-Heap** is like a memory factory.
- 📦 **DMA-BUF** is like a special ID tag that lets **anyone who supports it** (GPU, camera, display) pick up and use that memory.

---

## 🔄 How do they work together?

### 1. DMA-Heap allocates buffer

```c
int fd = open("/dev/dma_heap/system", O_RDWR);
int handle = ioctl(fd, DMA_HEAP_IOCTL_ALLOC, ...);  // get buffer
```

### 2. You get a DMA-BUF file descriptor

- That handle is a **DMA-BUF fd**
- You can pass it to GPU, display, etc. via `ioctl`, `mmap`, or Binder

---

## 🧩 Why do we need both?

| Need | Solution |
| --- | --- |
| Need to allocate memory from various physical regions (system, secure, carveout) | **DMA-Heap** |

| Need | Solution |
|---|---|
| Need to share that memory safely between hardware blocks (like GPU ↔ Display) | **DMA-BUF** |

---

# ✅ Summary

| Term | Simple Meaning | Think of it as... |
|---|---|---|
| DMA-Heap | Memory allocator | Factory making boxes |
| DMA-BUF | Shareable memory tag | Shipping label for that box |
| Used for | Camera, GPU, video | To move data between subsystems |

---

Let me know if you want a flow diagram or example in AOSP/Android context!

Trace window transitions using Winscope

bookmark_border
Winscope is a web tool that lets users record, replay, and analyze the states of several system services during and after animations and transitions. Winscope records all pertinent system service states to a trace file. Using the Winscope UI with the trace file, you can inspect the state of these services for each animation frame, with or without a screen recording, by replaying, stepping through, and debugging the transitions.

Supported traces
Winscope provides the ability to gather and visually represent various traces or sequences of system service states. You can configure these traces to suit specific use cases, ranging from low overhead to high verbosity. The following traces are supported by Winscope:

EventLog: Collect the system diagnostic event record using EventLog. In Winscope this information is used only to identify and display CUJ markings.
IME: Trace events from the Input Method Editor (IME) pipeline, including IMS, IMMS, and IME Client.
Input: Trace input events from various parts of the input event pipeline.
ProtoLog: Collect ProtoLog messages from system services and the code of system services running in client processes.
Screen recording: Collect a screen recording alongside the traces.
Shell transitions: Record window and activity transition system details.
SurfaceFlinger: Collect SurfaceFlinger traces containing information about surfaces (layers) such as position, buffer, and composition.
Transactions: Trace the set of atomic changes received by SurfaceFlinger using SurfaceControl for composition.
ViewCapture: Capture a range of properties of all the views from system Windows that support

ViewCapture, like System UI and Launcher.
Window Manager: Trace Window Manager states containing details pertaining to windows, including input and focus events, screen orientation, transitions, animations, positioning, and transformations.
Tip: The navigation timeline is shown in Winscope only if there is at least one trace. Each trace is represented in a separate row and each dot in the row represents a snapshot (state).
Supported dumps
Winscope can collect and display state dumps, which are snapshots of the device's state taken at specific moments defined by the user. Unlike traces, which are continuously collected during device usage and can impact performance, dumps are taken only at these user-defined moments, ensuring that performance and verbosity are not compromised. This allows for a more focused and efficient analysis of the device's state at specific points in time. The following dumps are supported by Winscope:

Window Manager: Dump a single Window Manager state.
SurfaceFlinger: Dump a single SurfaceFlinger snapshot.
Screenshot: Collect a screenshot alongside the dumps.
Tip: The navigation timeline is shown in Winscope only if there is at least one trace. If there's a combination of traces and dumps, the timeline will show a row with a single entry (dot) to represent a dump. Moreover, tabs containing dumps are suffixed with the word Dump, for example, SurfaceFlinger Dump.
Resources
See Run Winscope for information on building and running Winscope.

See Capture traces for information on collecting traces.

See Load traces for information on how to load traces using the Winscope Web UI.

See Analyze traces for information on analyzing traces.

Examples
The following example describe how to debug a flicker test failure and a user reported bug.

Flicker test failure
This example demonstrates how to use Winscope to debug a flicker test failure.

Examine the test failure
Follow these steps to determine the issue type and examine the test failure message.

Determine the issue type by examining the test and class name.

Test and class name:

FlickerTestsNotification
com.android.server.wm.flicker.notification.OpenAppFromLockscreenNotificationColdTest#appL

ayerBecomesVisible[ROTATION_0_GESTURAL_NAV]
Issue type:

The CUJ refers to launching an app from a lockscreen notification (OpenAppFromLockscreenNotificationColdTest).

The test expects the app to become visible (#appLayerBecomesVisible).

Examine the test failure message, which provides comprehensive information about the failure, including:

A comparison between the expected outcome and the actual visible outcome
Timestamps to help pinpoint when the failure occurred
The name of the artifact or file associated with the failure
Additional contextual information relevant to understanding and debugging the failure

android.tools.flicker.subject.exceptions.IncorrectVisibilityException:
com.android.server.wm.flicker.testapp/com.android.server.wm.flicker.testapp.NotificationActivity# should be visible

Where?
    Timestamp(UNIX=2024-05-10T11:04:14.227572545(1715339054227572545ns), UPTIME=37m21s184ms79178ns(2241184079178ns), ELAPSED=0ns)

What?
    Expected:
com.android.server.wm.flicker.testapp/com.android.server.wm.flicker.testapp.NotificationActivity#
    Actual: [e636ecd
com.android.server.wm.flicker.testapp/com.android.server.wm.flicker.testapp.NotificationActivity#3457: Buffer is empty, Visible region calculated by Composition Engine is empty,
com.android.server.wm.flicker.testapp/com.android.server.wm.flicker.testapp.NotificationActivity#3458: Visible region calculated by Composition Engine is empty]

Other information
    Artifact:
FAIL__OpenAppFromLockscreenNotificationColdTest_ROTATION_0_GESTURAL_NAV.zip

Check the test run artifacts for trace files

    at
android.tools.flicker.subject.layers.LayerTraceEntrySubject.isVisible(LayerTraceEntrySubject.kt:187)
    at
android.tools.flicker.subject.layers.LayersTraceSubject$isVisible$1$1.invoke(LayersTraceSubject.kt:151)

```
    at
android.tools.flicker.subject.layers.LayersTraceSubject$isVisible$1$1.invoke(LayersTraceSubject.kt:150)
    at android.tools.flicker.assertions.NamedAssertion.invoke(NamedAssertion.kt:32)
    at android.tools.flicker.assertions.CompoundAssertion.invoke(CompoundAssertion.kt:42)
    at android.tools.flicker.assertions.AssertionsChecker.test(AssertionsChecker.kt:79)
    at android.tools.flicker.subject.FlickerTraceSubject.forAllEntries(FlickerTraceSubject.kt:59)
    at
android.tools.flicker.assertions.AssertionDataFactory$createTraceAssertion$closedAssertion$1.invoke(AssertionDataFactory.kt:46)
    at
android.tools.flicker.assertions.AssertionDataFactory$createTraceAssertion$closedAssertion$1.invoke(AssertionDataFactory.kt:43)
    at android.tools.flicker.assertions.AssertionDataImpl.checkAssertion(AssertionDataImpl.kt:33)
    at
android.tools.flicker.assertions.ReaderAssertionRunner.doRunAssertion(ReaderAssertionRunner.kt:35)
    at
android.tools.flicker.assertions.ReaderAssertionRunner.runAssertion(ReaderAssertionRunner.kt:29)
    at
android.tools.flicker.assertions.BaseAssertionRunner.runAssertion(BaseAssertionRunner.kt:36)
    at android.tools.flicker.legacy.LegacyFlickerTest.doProcess(LegacyFlickerTest.kt:59)
    at android.tools.flicker.assertions.BaseFlickerTest.assertLayers(BaseFlickerTest.kt:89)
    at
com.android.server.wm.flicker.notification.OpenAppTransition.appLayerBecomesVisible_coldStart(OpenAppTransition.kt:51)
    at
com.android.server.wm.flicker.notification.OpenAppFromNotificationColdTest.appLayerBecomesVisible(OpenAppFromNotificationColdTest.kt:64)
This output sample indicates the following:

The issue happens at 2024-05-10T11:04:14.227572545.

NotificationActivity is expected to be visible but it isn't.

The name of the artifact file that contains the traces for debugging is
FAIL__OpenAppFromLockscreenNotificationColdTest_ROTATION_0_GESTURAL_NAV.
```

Here's a **simple summary** of what **Winscope** is and how you can use it to **trace and debug window transitions** like app launches, flickers, or animations in Android:

---

## 🧰 What is Winscope?

**Winscope** is a **web-based visualization tool** for Android developers and engineers.
It helps you **record, replay, and analyze** system-level events during **UI transitions** (like launching apps, switching screens, opening keyboard, etc.).

---

## 🎯 Why use Winscope?

It helps you:

- Debug **UI glitches**, **slow animations**, or **unexpected flickers**
- Understand what's happening at every frame of an animation
- Inspect state from multiple system services like **SurfaceFlinger**, **Window Manager**, **Input**, etc.

---

## 📦 What does Winscope collect?

### 🔁 Traces (Continuous Recording)

These are **time-based recordings** of events:

| Trace Type | What it Records |
| --- | --- |
| **SurfaceFlinger** | Layers, buffers, composition |
| **Window Manager** | Windows, input focus, animations |
| **Input** | Touch and input events |
| **Shell Transitions** | System transitions |
| **IME** | Keyboard and IME states |
| **Screen Recording** | Optional video to match events |
| **Transactions** | Buffer updates and atomic operations |
| **ViewCapture** | UI view properties in SystemUI/Launcher |
| **ProtoLog** | Internal logs |
| **EventLog** | CUJ markers (Common User Journeys) |

---

### 📸 Dumps (One-time Snapshots)

These are taken at **specific moments**, not continuously:

| Dump Type | Description |
| --- | --- |
| **Window Manager Dump** | Current state of windows |
| **SurfaceFlinger Dump** | Current layer and buffer states |
| **Screenshot** | Visual capture of that moment |

---

# 🔍 How does Winscope help in debugging?

### Example: App flicker during notification launch

1. **Test failed**: App should've become visible, but didn't.
2. **Winscope Trace** shows:
    - The **layer was created**, but the **buffer was empty**.
    - **Visible region = empty**, so app was not shown.
3. **Timeline navigation** lets you:
    - Jump to the exact **timestamp of failure**
    - View SurfaceFlinger + WindowManager side-by-side
    - Check input events, transitions, and layer visibility

---

# 🛠️ How to use Winscope?

### 1. Capture traces

- Use tools like `flicker` or `atrace` to collect logs
- Add screen recording if needed

### 2. Open Winscope

- Visit: https://winscope.dev or run locally
- Load `.zip` or `.proto` trace files

### 3. Analyze

- Step through each frame of animation
- Look at **why layers were not visible**
- Match behavior to code expectations

---

# ✅ Key Benefits

- Combines **multiple system traces** into a single UI
- Helps trace complex bugs like:
    - Flickering UI
    - Keyboard not appearing

- App transitions failing
- Missing animations
- Useful for both **manual debugging** and **automated test validation**

---

If you want, I can show you:

- A diagram of the Winscope layers
- How to run a simple test and load the trace
- How to interpret a sample trace file

Let me know!

how to access winscope

You can access **Winscope** in two ways — through the **official website** or by **running it locally**. Here's how:

---

# ✅ 1. Access via Website (Most Common)

### 🌐 Go to:

👉 https://winscope.dev

### 🧾 Steps:

1. Visit the website.
2. Click **"Open Trace"** or **drag-and-drop** a trace file (usually `.zip`, `.proto`, or `.trace`).
3. Start exploring transitions, layers, and window states.

> 📌 **No installation needed**. Just open the site in Chrome or Chromium-based browsers for best results.

---

# 🧰 2. Run Winscope Locally (Optional, for advanced usage)

### 🧑‍💻 Requirements:

- **Node.js + npm**
- Git (to clone the repo)

## 🛠️ Steps:

```bash
git clone https://github.com/google/winscope.git
cd winscope
npm install
npm run start
```

Then open http://localhost:8080 in your browser.

---

# 📥 How to Get a Trace File

Use the **Flicker test framework** or run this ADB command on your Android device:

```bash
adb shell cmd window trace start
# Reproduce your bug here...
adb shell cmd window trace stop
adb pull /data/misc/wmtrace/wm_trace.winscope
```

Other common trace sources:

- **`SurfaceFlinger`** trace:
  **`adb shell dumpsys SurfaceFlinger --proto | tee sf_trace.pb`**
- **`WindowManager`** trace:
  **`adb shell dumpsys window --proto | tee wm_trace.pb`**

Then open them on the Winscope website.

---

Would you like a step-by-step example of capturing a trace and viewing it?

yes

Here's a **step-by-step guide** to **capture a trace** from an Android device and **view it in Winscope**.

---

# 🐾 Step-by-Step: Capture & View Trace in Winscope

---

## ✅ Step 1: Enable Developer Options

On your Android device:

1. Go to **Settings > About phone**
2. Tap **Build number** 7 times to enable developer options.
3. Go back to **Settings > System > Developer options**
4. Enable **USB debugging**

---

## ✅ Step 2: Connect Device via ADB

On your PC:

```bash
adb devices
```

Make sure your device shows up. If not, authorize the USB prompt on your phone.

---

## ✅ Step 3: Start a WindowManager Trace

```bash
adb shell cmd window trace start
```

This starts capturing **WindowManager** activity.

---

## ✅ Step 4: Trigger the Action You Want to Trace

Do something on your device — open an app, rotate screen, launch keyboard, etc.

---

## ✅ Step 5: Stop the Trace

```bash
adb shell cmd window trace stop
```

---

## ✅ Step 6: Pull the Trace File to Your PC

```bash
adb pull /data/misc/wmtrace/wm_trace.winscope
```

You now have a file like: `wm_trace.winscope`

---

### ✅ Step 7: Open Winscope in Browser

Go to:

👉 https://winscope.dev

Then:

- Click **"Open Trace"**
- Select the `wm_trace.winscope` file
- Explore the captured window and layout changes

---

## 🧠 Want SurfaceFlinger or Layer Traces Too?

```bash
# SurfaceFlinger trace
adb shell dumpsys SurfaceFlinger --proto > sf_trace.pb

# WindowManager trace
adb shell dumpsys window --proto > wm_trace.pb
```

Then upload both `.pb` files in Winscope together to see a **merged view** of window + layers + transitions.

---

Let me know if you want help interpreting the trace or automating this via script.

Run Winscope

bookmark_border
Winscope tracing is part of the Android framework. This page outlines the steps required to download, build, and run the Winscope trace viewer locally.

Build Winscope locally
Follow these steps to set up your PC to run the Winscope tracer:

Download the Android source.
Navigate to the Winscope folder:

cd development/tools/winscope
Install dependencies using:

npm install
To see a list of available commands, run: npm run

Build all prod and test targets using:

npm run build:prod
Run Winscope using:

npm run start
Note: You can also download and build Winscope to host on a standard web server to get a shareable link to the tool.
Build separate parts
You can build individual parts of Winscope separately using the following commands:

Command Description
build:trace_processor Checks out and rebuilds the latest version of
Perfetto's trace_processor.
build:protos Recompiles the proto definitions.
Run tests
Tip: Winscope doesn't build on Soong and can't be built by Tradefed, so it's recommended to have a local Git hook to the tests in your local environment.
Winscope contains unit and end-to-end tests. To run them use npm run <command>:

Command Description
test:unit:ci Runs the unit tests in a less verbose format for CI
or presubmit hook.
test:unit:dev Runs the unit tests in a more verbose format for
local development. This mode watches for changes and
automatically reruns the correct tests.
test:e2e Runs the end-to-end tests, such as those for the
cross-tool protocol.
test:presubmit:quiet Builds all presubmit unit tests, linters, and graph
analysis in a less verbose format for CI or presubmit
hook.
test:presubmit Builds all presubmit unit tests, linters, and graph
analysis in a more verbose format for local
development.
test:all Runs all tests (unit and end-to-end), linters, and
graph analysis in a more verbose format for local
development.
Update @IntDef mapping
@IntDef is an annotation used in Android to restrict the possible values of an integer. Winscope

uses a mapping of these annotations to display the name of the value instead of the integer.

To update the @IntDef mapping, do the following:

Build :framework-minus-apex-intdefs for the annotation preprocessor to run:

mp :framework-minus-apex-intdefs
Copy the generated intDefMapping.json file to the prebuilts repository:

$ python3 -c 'import sys,json,collections;
print(json.dumps(collections.OrderedDict(sorted(collections.ChainMap(*map(lambda
x:json.load(open(x)), sys.argv[1:])).items())), indent=2))' $(find
out/soong/.intermediates/frameworks/base -iname intDefMapping.json) >
./development/tools/winscope/src/common/intDefMapping.json
Upload the changes in Winscope using repo upload.

Other commands
In addition to build and tests, Winscope scripts contain other capabilities, as shown in the table.
To run them use npm run command:

Command Description
format:check Checks for code formatting issues using prettier.
format:fix Checks and auto fixes code formatting issues using prettier.
eslint:check Checks for code formatting issues using eslint.
eslint:fix Checks and auto fixes code formatting issues using eslint.
tslint:check Checks for code formatting issues using tslint.
tslint:fix Checks and auto fixes code formatting issues using tslint.
deps_graph:check_cycles Analyzes the code for cyclical dependencies.
Troubleshoot
Use the following tips for troubleshooting:

Error ProtocolError: missing required '<FIELD>' or TypeError: Cannot read property '<PROP>' of null

This occurs when the trace file was created with a new proto definition, containing new required fields.

Make sure you're opening the trace on the correct Winscope version (master, S, or R).
If you create the new field on the proto, recompile the protos in Winscope using npm run build:protos.

Note: You shouldn't need a new required field, because the protos should be backward compatible.
Some installed dependency versions are wrong (build fails)

Revert changes to package.json and package-lock.json. Remove node_modules. Run npm install again.
I added a new field to one of the proto files. How do I display it?

Winscope uses the proto definitions from when it was compiled, so new fields won't appear by default. To show new fields, rebuild the protos using npm run build:protos.

Here's a **simple step-by-step guide** to help you **access and run Winscope** locally on your PC:

---

## ✅ What is Winscope?

Winscope is a tool to **visualize Android system traces**, like **WindowManager**, **SurfaceFlinger**, **IME**, and **Transition animations**, to help debug display and windowing issues.

---

## 🧰 Requirements:

- You must have downloaded the **AOSP source code**
- Node.js and npm installed (version ≥ 16 recommended)
- Linux or macOS terminal (for commands below)

---

## 🧪 Steps to Run Winscope Locally

1. **Go to the Winscope directory in AOSP:**

```bash
cd development/tools/winscope
```

2. **Install all required packages:**

```bash
npm install
```

3. **Build Winscope (production version):**

```bash
npm run build:prod
```

4. **Start Winscope in browser:**

```bash
npm run start
```

- This will open http://localhost:5000 in your browser.

---

## 📁 If You Want to Share Winscope on a Web Server

You can copy the **`dist/`** folder contents and host it on any standard web server (like Nginx, Apache, or GitHub Pages).

---

## ✅ To View a Trace in Winscope

1. On your Android device:

```bash
adb shell cmd window trace start
# do your interaction
adb shell cmd window trace stop
adb pull /data/misc/wmtrace/wm_trace.winscope
```

2. In the Winscope browser window, load the **`.winscope`** file.

---

## 🛠️ Common Commands

| Command | What It Does |
|---|---|
| **`npm run build:prod`** | Builds the production-ready Winscope |
| **`npm run build:protos`** | Recompiles trace proto files |
| **`npm run test:unit:dev`** | Runs unit tests (for dev) |
| **`npm run format:fix`** | Fixes formatting with Prettier |
| **`npm run eslint:fix`** | Fixes JS/TS lint errors |

---

## 🧯 Troubleshooting Tips

- **Build Fails?**
  - Try:

```bash
rm -rf node_modules package-lock.json
npm install
```

- **Missing trace field or crash when loading trace?**
  - Your trace might be from a newer Android version than your Winscope build.
  - Fix: run `npm run build:protos` to update proto definitions.

---

Would you like a script to **automate these steps** on your machine?

Surface and SurfaceHolder

bookmark_border
Surface objects enable apps to render images to be presented on screens. SurfaceHolder interfaces enable apps to edit and control surfaces.

Surface
A surface is an interface for a producer to exchange buffers with a consumer.

The BufferQueue for a display surface is typically configured for triple-buffering. Buffers are allocated on demand, so if the producer generates buffers slowly enough, such as at 30 fps on a 60 fps display, there might only be two allocated buffers in the queue. Allocating buffers on demand helps minimize memory consumption. You can see a summary of the buffers associated with every layer in the dumpsys SurfaceFlinger output.

Most clients render onto surfaces using OpenGL ES or Vulkan. However, some clients render onto surfaces using a canvas.

Canvas rendering
The canvas implementation is provided by the Skia Graphics Library. If you want to draw a rectangle, you call the Canvas API, which sets bytes in a buffer appropriately. To ensure that a buffer isn't updated by two clients at once, or written to while being displayed, lock the buffer to access it. Use the following commands to work with canvas locks:

lockCanvas() locks the buffer for rendering on the CPU and returns a Canvas to use for drawing.
unlockCanvasAndPost() unlocks the buffer and sends it to the compositor.
lockHardwareCanvas() locks the buffer for rendering on the GPU and returns a canvas to use for drawing.
Note: The canvas obtained when an app locks a surface with lockCanvas() is never hardware accelerated.
Caution: You can't draw on a surface with GLES or send it frames from a video decoder if you've ever called lockCanvas(). lockCanvas() connects the CPU renderer to the producer side of the BufferQueue and doesn't disconnect until the surface is destroyed. The canvas-based CPU renderer can't be disconnected and reconnected to a surface, unlike most producers (like GLES or Vulkan).

The first time the producer requests a buffer from a BufferQueue, the buffer is allocated and initialized to zero. Initialization is necessary to avoid inadvertently sharing data between processes. However, if you reuse a buffer, the previous contents are still present. If you repeatedly call lockCanvas() and unlockCanvasAndPost() without drawing anything, the producer cycles between previously rendered frames.

The surface lock/unlock code keeps a reference to the previously rendered buffer. If you specify a dirty region when locking the surface, it copies the nondirty pixels from the previous buffer. SurfaceFlinger or HWC typically handle the buffer; but because we only need to read from the buffer, there's no need to wait for exclusive access.

SurfaceHolder
A SurfaceHolder is an interface the system uses to share ownership of surfaces with apps. Some clients that work with surfaces want a SurfaceHolder, because APIs to get and set surface parameters are implemented through a SurfaceHolder. A SurfaceView contains a SurfaceHolder.

Most components that interact with a view involve a SurfaceHolder. Some other APIs, such as MediaCodec, operate on the surface itself.

W

Here’s a simple breakdown of the **relationship between `Surface`, `SurfaceHolder`, `SurfaceView`, and `BufferQueue`** based on the information you shared and what we’ve discussed earlier:

---

## ✅ 1. Surface

- A `Surface` is **a handle to a producer side** of a **BufferQueue**.
- It's the main thing you render onto using:
  - **OpenGL ES**
  - **Vulkan**
  - **MediaCodec**
  - **Canvas (software)**

It **acts as a producer**, pushing frames into the **BufferQueue**, which is then read by a **consumer** (like **SurfaceFlinger**, the compositor).

---

## ✅ 2. SurfaceHolder

- A **helper interface** to manage and configure a `Surface`.
- Used **with `SurfaceView`**.

- Allows access to:
  - The underlying **`Surface`**
  - Surface lifecycle methods (**`surfaceCreated`**, **`surfaceDestroyed`**, etc.)
  - Buffer properties (size, format)

Think of it like a **manager** that helps the app talk to the **`Surface`** safely.

---

## ✅ 3. SurfaceView

- A **View subclass** (part of the Android UI) that **creates a separate drawing surface**.
- Internally holds a **`SurfaceHolder`**, which holds a **`Surface`**.
- Used when rendering needs to happen **outside of the main UI thread**, like:
  - Video playback
  - OpenGL rendering
  - Camera preview

⚠️ Unlike regular Views, it **doesn't draw with the View system**, but through its own surface (layer).

---

## ✅ 4. BufferQueue

- A communication **channel between producer and consumer**.
- Typically:
  - **Producer**: App (**`Surface`**, **`SurfaceView`**, **`MediaCodec`**, etc.)
  - **Consumer**: **`SurfaceFlinger`**, **`Hardware Composer`**, **`Camera HAL`**, etc.

A **`BufferQueue`** manages:

- **Buffer allocation**
- **Buffer state transitions** (free → dequeued → queued → acquired → released)

---

## ✅ How they connect:

| Component | Role | Talks to |
|---|---|---|
| **`SurfaceView`** | View for separate rendering | Uses **`SurfaceHolder`** |
| **`SurfaceHolder`** | Manages the Surface | Gives access to **`Surface`** |
| **`Surface`** | Buffer producer interface | Produces to **`BufferQueue`** |
| **`BufferQueue`** | Shared memory queue | Passes buffers to consumer like **`SurfaceFlinger`** |

---

## 🧠 Example flow:

1. App uses a **`SurfaceView`** to show video.
2. **`SurfaceView`** has a **`SurfaceHolder`**.
3. **`SurfaceHolder`** exposes a **`Surface`**.
4. App draws onto **`Surface`** using OpenGL/Canvas.
5. **`Surface`** is a producer into a **`BufferQueue`**.
6. **`SurfaceFlinger`** is the consumer and composites the final image to the screen.

---

Let me know if you'd like a diagram to visualize this or if any part is still unclear.

SurfaceView and GLSurfaceView

bookmark_border
The Android app framework UI is based on a hierarchy of objects that start with a View. All UI elements go through a series of measurements and a layout process that fits them into a rectangular area. Then, all visible view objects are rendered to a surface that was set up by the WindowManager when the app was brought to the foreground. The app's UI thread performs layout and rendering to a buffer per frame.

SurfaceView
A SurfaceView is a component that you can use to embed an additional composite layer within your view hierarchy. A SurfaceView takes the same layout parameters as other views, so it can be manipulated like any other view, but the SurfaceView's contents are transparent.

When you render with an external buffer source, such as GL context or a media decoder, you need to copy buffers from the buffer source to display the buffers on the screen. Using a SurfaceView enables you to do that.

When the SurfaceView's view component is about to become visible, the framework asks SurfaceControl to request a new surface from SurfaceFlinger. To receive callbacks when the surface is created or destroyed, use the SurfaceHolder interface. By default, the newly created surface is placed behind the app UI surface. You can override the default Z-ordering to put the new surface on top.

Rendering with SurfaceView is beneficial in cases where you need to render to a separate surface, such as when you render with the Camera API or an OpenGL ES context. When you render with SurfaceView, SurfaceFlinger directly composes buffers to the screen. Without a SurfaceView, you need to composite buffers to an offscreen surface, which then gets composited to the screen, so rendering with SurfaceView eliminates extra work. After rendering with SurfaceView, use the UI thread to coordinate with the activity lifecycle and make adjustments to the size or position of the view if needed. Then, the Hardware Composer blends the app UI and the other layers.

The new surface is the producer side of a BufferQueue, whose consumer is a SurfaceFlinger layer. You can update the surface with any mechanism that can feed a BufferQueue, such as surface-supplied Canvas functions, attaching an EGLSurface and drawing on the surface with GLES, or configuring a media decoder to write the surface.

SurfaceView and the activity lifecycle
When using a SurfaceView, render the surface from a thread other than the main UI thread.

For an activity with a SurfaceView, there are two separate but interdependent state machines:

App onCreate/onResume/onPause
Surface created/changed/destroyed
When the activity starts, you get callbacks in this order:

onCreate()
onResume()
surfaceCreated()
surfaceChanged()
If you click back, you get:

onPause()
surfaceDestroyed() (called just before the surface goes away)
If you rotate the screen, the activity is torn down and recreated and you get the full cycle. You can tell it's a quick restart by checking isFinishing(). It's possible to start/stop an activity so quickly that surfaceCreated() happens after onPause().

If you tap the power button to blank the screen, you get only onPause() without surfaceDestroyed(). The surface remains active, and rendering can continue. You can keep getting Choreographer events if you continue to request them. If you have a lock screen that forces a different orientation, your activity may be restarted when the device is unblanked. Otherwise, you can come out of screen-blank with the same surface as before.

The lifespan of the thread can be tied to the surface or to the activity, depending on what you want to happen when the screen goes blank. The thread can start/stop either on Activity start/stop or on surface create/destroy.

Having the thread start/stop on Activity start/stop works well with the app lifecycle. You start the renderer thread in onResume() and stop it in onStop(). When creating and configuring the thread, sometimes the surface already exists, othertimes it doesn't (for example, it's still active after toggling the screen with the power button). You have to wait for the surface to be created before initializing in the thread. You can't initialize in the surfaceCreate() callback because it won't fire again if the surface wasn't recreated. Instead, query or cache the surface state, and forward it to the renderer thread.

Note: Be careful when passing objects between threads. It is best to pass the surface or SurfaceHolder through a Handler message (rather than just stuffing it into the thread) to avoid issues on multicore systems. For details, refer to SMP Primer for Android.
Having the thread start/stop on surface create/destroy works well because the surface and the renderer are logically intertwined. You start the thread after the surface is created, which avoids some interthread communication concerns; and surface created/changed messages are simply forwarded. To ensure that rendering stops when the screen goes blank and resumes when it un-blanks, tell Choreographer to stop invoking the frame draw callback. onResume() resumes the callbacks if the renderer thread is running. However, if you animate based on elapsed time between frames, there could be a large gap before the next event arrives; using an explicit pause/resume message can solve this issue.

Note: For an example of having a thread start/stop on surface create/destroy, see Grafika's Hardware scaler activity.
Both options, whether the lifespan of the thread is tied to the Activity or the surface, focus on how the renderer thread is configured and whether it's executing. A related concern is extracting state from the thread when the activity is killed (in onStop() or onSaveInstanceState()); in such cases, tying the lifespan of the thread to the activity works best because after the renderer thread has been joined, the rendered thread's state can be accessed without synchronization primitives.

GLSurfaceView
The GLSurfaceView class provides helper classes for managing EGL contexts, interthread communication, and interaction with the activity lifecycle. You don't need to use a GLSurfaceView to use GLES.

For example, GLSurfaceView creates a thread for rendering and configures an EGL context there. The state is cleaned up automatically when the activity pauses. Most apps don't need to know anything about EGL to use GLES with GLSurfaceView.

In most cases, GLSurfaceView can make working with GLES easier. In some situations, it can get in the way.

Here's a **simple comparison and relationship between `SurfaceView` and `GLSurfaceView`**:

---

## ✅ SurfaceView

- **Purpose**: To render onto a **separate Surface**, bypassing the normal View rendering pipeline.
- **Used for**:
  - Camera previews
  - Video playback
  - OpenGL rendering

- Media decoding output

🔧 **Key Characteristics:**

- Hosts a **`Surface`** that you can draw onto from a **separate thread**.
- App provides the rendering logic (thread creation, EGL setup, etc.).
- Gives **maximum control**, but **requires more manual work**.
- Surface is backed by a **BufferQueue**, consumed by SurfaceFlinger directly.
- Managed through a **`SurfaceHolder`**.

---

## ✅ GLSurfaceView

- **Purpose**: A subclass of **`SurfaceView`** specifically designed for **OpenGL ES rendering**.
- **Used for**:
  - Games
  - 3D rendering
  - Visualizations with OpenGL

🔧 **Key Characteristics:**

- Simplifies:
  - EGL context setup
  - Rendering thread management
  - Activity lifecycle integration
- You provide a **`Renderer`** implementation (a class with **`onDrawFrame()`**, **`onSurfaceCreated()`**, etc.).
- **Handles threading, context loss, pause/resume** for you.
- Great for **OpenGL beginners or standard rendering needs**.

---

## 📊 Summary Table

| Feature | `SurfaceView` | `GLSurfaceView` |
|---|---|---|
| Purpose | General-purpose external rendering | Simplified OpenGL ES rendering |
| Backed by | **`Surface`** via **`SurfaceHolder`** | **`Surface`** via **`SurfaceHolder`** |
| Rendering Thread | You manage manually | Managed by **`GLSurfaceView`** internally |
| EGL Context | You create and manage | Created and managed internally |
| Activity Lifecycle | You handle manually | Handled for you |

| Feature | `SurfaceView` | `GLSurfaceView` |
|---|---|---|
| Use Cases | Camera, Video, MediaCodec | OpenGL-based games and 3D rendering |
| Customization | High (manual control) | Moderate (easier but more restricted) |

---

## 🧠 Use Case Guidance

| If you want to... | Use... |
|---|---|
| Play video from MediaCodec | `SurfaceView` |
| Render camera preview | `SurfaceView` |
| Render 3D graphics or OpenGL scenes | `GLSurfaceView` |
| Handle all rendering logic manually with fine control | `SurfaceView` |
| Get started with OpenGL quickly | `GLSurfaceView` |

---

Let me know if you'd like code examples for either approach!

SurfaceTexture

bookmark_border
SurfaceTexture is a combination of a surface and an OpenGL ES (GLES) texture. SurfaceTexture instances are used to provide surfaces that output to GLES textures.

SurfaceTexture contains an instance of BufferQueue for which apps are the consumer. The onFrameAvailable() callback notifies apps when the producer queues a new buffer. Then, apps call updateTexImage(), which releases the previously held buffer, acquires the new buffer from the queue, and makes EGL calls to make the buffer available to GLES as an external texture.

External GLES textures
External GLES textures (GL_TEXTURE_EXTERNAL_OES) differ from traditional GLES textures (GL_TEXTURE_2D) in the following ways:

External textures render textured polygons directly from data received from BufferQueue.
External texture renderers are configured differently than traditional GLES texture renderers.
External textures can't perform all traditional GLES texture activities.
The main benefit of external textures is their ability to render directly from BufferQueue data.
SurfaceTexture instances set the consumer usage flags to GRALLOC_USAGE_HW_TEXTURE when it creates BufferQueue instances for external textures to ensure that the data in the buffer is recognizable by GLES.

Because SurfaceTexture instances interact with an EGL context, an app can only call its methods while the EGL context that owns the texture is current on the calling thread. For more information see the SurfaceTexture class documentation.

Timestamps and transformations
SurfaceTexture instances include the getTimeStamp() method, which retrieves a timestamp, and getTransformMatrix() method, which retrieves a transformation matrix. Calling updateTexImage() sets both the timestamp and the transformation matrix. Each buffer that BufferQueue passes includes transformation parameters and a timestamp.

Transformation parameters are useful for efficiency. In some cases, source data might be in the incorrect orientation for the consumer. Instead of rotating the data before sending it to the consumer, send the data in its orientation with a transform that corrects it. The transformation matrix can be merged with other transformations when the data is used, minimizing overhead.

The timestamp is useful for buffer sources that are time dependent. For example, when setPreviewTexture() connects the producer interface to the output of the camera, frames from the camera can be used to create a video. Each frame needs to have a presentation timestamp from when the frame was captured, not from when the app received the frame. The camera code sets the timestamp provided with the buffer, resulting in a more consistent series of timestamps.

Case study: Grafika's continuous capture
Grafika's continuous capture involves recording frames from a device's camera and displaying those frames on screen. To record frames, create a surface with the MediaCodec class's createInputSurface() method and pass the surface to the camera. To display frames, create an instance of SurfaceView and pass the surface to setPreviewDisplay(). Note that recording frames and displaying them at the same time is a more involved process.

The continuous capture activity displays video from the camera as video is being recorded. In this case, encoded video is written to a circular buffer in memory that can be saved to disk at any time.

This flow involves three buffer queues:

App — The app uses a SurfaceTexture instance to receive frames from the camera, converting them to an external GLES texture.
SurfaceFlinger — The app declares a SurfaceView instance to display the frames.
MediaServer — Configure a MediaCodec encoder with an input surface to create the video.
In the figure below, the arrows indicate data propagation from the camera. BufferQueue instances are in color (producers are teal, consumers are green).

Grafika continuous
capture activity
Figure 1. Grafika's continuous capture activity

Encoded H.264 video goes to a circular buffer in RAM in the app process. When a user presses the capture button, the MediaMuxer class writes the encoded video to an MP4 file on disk.

All BufferQueue instances are handled with a single EGL context in the app while the GLES operations are performed on the UI thread. The handling of encoded data (managing a circular buffer and writing it to disk) is done on a separate thread.

Warning: If the video encoder locks up and blocks a dequeueing buffer the app becomes unresponsive.
Note: Performing SurfaceView rendering on the UI thread is discouraged, but it's used in this case study for simplicity. More complex rendering should use a dedicated thread to isolate the GLES context and minimize interference with the rendering of another app's UI.
When using the SurfaceView class, the surfaceCreated() callback creates the EGLContext and EGLSurface instances for the display and the video encoder. When a new frame arrives, SurfaceTexture performs four activities:
Acquires the frame.
Makes the frame available as a GLES texture.
Renders the frame with GLES commands.
Forwards the transform and timestamp for each instance of EGLSurface.
The encoder thread then pulls the encoded output from MediaCodec and stashes it in memory.

Secure texture video playback
Android supports GPU post-processing of protected video content. This lets apps use the GPU for complex, nonlinear video effects (such as warps), mapping protected video content onto textures for use in general graphics scenes (for example, using GLES), and virtual reality (VR).

Secure Texture Video Playback
Figure 2. Secure texture video playback

Support is enabled using the following two extensions:

EGL extension — (EGL_EXT_protected_content) Enables the creation of protected GL contexts and surfaces, which can both operate on protected content.
GLES extension — (GL_EXT_protected_textures) Enables tagging textures as protected so they can be used as framebuffer texture attachments.
Android enables SurfaceTexture and ACodec (libstagefright.so) to send protected content even if the window's surface doesn't queue to SurfaceFlinger and provides a protected video surface for use within a protected context. This is done by setting the protected consumer bit (GRALLOC_USAGE_PROTECTED) on surfaces created in a protected context (verified by ACodec).

Secure texture video playback sets the foundation for strong DRM implementation in the OpenGL ES environment. Without a strong DRM implementation, such as Widevine Level 1, many content providers don't allow rendering of their high-value content in the OpenGL ES environment, preventing important VR use cases such as watching DRM-protected content in VR.

AOSP includes framework code for secure texture video playback. Driver support is up to OEMs. Device implementers must implement EGL_EXT_protected_content and

GL_EXT_protected_textures extensions. When using your own codec library (to replace libstagefright), note the changes in /frameworks/av/media/libstagefright/SurfaceUtils.cpp that allow buffers marked with GRALLOC_USAGE_PROTECTED to be sent to ANativeWindow (even if ANativeWindow doesn't queue directly to the window composer) as long as the consumer usage bits contain GRALLOC_USAGE_PROTECTED. For detailed documentation on implementing the extensions, refer to the Khronos registries (EGL_EXT_protected_content, and GL_EXT_protected_textures).

Note: Device implementers may need to make hardware changes to ensure that protected memory mapped onto the GPU remains protected and unreadable by unprotected code.

Here's a **simple and clear explanation of `SurfaceTexture`**, its **relationship to OpenGL**, and how it's **different from SurfaceView or Surface**.

---

## 🧱 What is `SurfaceTexture`?

- **`SurfaceTexture`** = **Surface + GLES texture**
- It's used to **receive image frames from a producer** (like a camera or video decoder) and **expose them as a texture to OpenGL ES**.
- Allows you to **draw camera frames or video frames directly onto 3D objects or surfaces using OpenGL.**

---

## 🔄 How it Works (BufferQueue-based)

- Internally backed by a **BufferQueue**.
- **App is the consumer**, **Producer could be**:
  - Camera
  - MediaCodec
  - Video decoder, etc.

### Flow:

1. Producer pushes a new frame → **`BufferQueue`**
2. App gets notified via **`onFrameAvailable()`** callback
3. App calls **`updateTexImage()`**:
   - Acquires latest frame
   - Makes it available as a **GL_TEXTURE_EXTERNAL_OES**
   - Updates timestamp + transform matrix

# 🖼️ External GLES Texture (`GL_TEXTURE_EXTERNAL_OES`)

- Special type of texture that **connects directly to buffer data**.
- Used for:
    - Live camera previews
    - Media playback
    - Augmented reality
- You **can't manipulate it like a normal `GL_TEXTURE_2D`**.

# 🕹️ Why Use `SurfaceTexture`?

| Feature | Benefit |
|---|---|
| Direct OpenGL rendering | Frame becomes a texture you can map onto 3D surfaces |
| Efficient GPU rendering | No CPU readback/conversion needed |
| Real-time video effects | Enables warping, filtering, or compositing |
| Camera preview | Use OpenGL to show the preview with effects |
| MediaCodec input/output | Used for encoding/decoding with OpenGL support |

# 🎯 Use Cases

1. **Camera Preview + 3D Effects**
    - Camera feeds `SurfaceTexture`
    - You render it as a live texture in OpenGL
    - Example: AR apps, camera filters
2. **Video Playback in OpenGL**
    - Use `MediaCodec` output surface as `SurfaceTexture`
    - Decode video directly to a GLES texture
3. **Secure Video Playback**
    - Uses **protected memory + GPU**
    - Ensures video can't be screen-recorded or read from memory
    - Important for DRM (e.g. Widevine L1, Netflix, Prime Video)

# 🔐 Secure Texture Playback (Optional)

- Uses **protected EGL contexts + protected memory buffers**.
- Prevents any app from reading the frame (for DRM-protected content).
- Required for **VR and DRM content playback** in apps.

---

## ⚙️ Summary Table

| Component | Role |
|---|---|
| `SurfaceTexture` | Gets buffers, exposes as GL texture |
| `GL_TEXTURE_EXTERNAL_OES` | Renders directly from BufferQueue |
| `updateTexImage()` | Pulls latest buffer + updates texture |
| `onFrameAvailable()` | Callback when new frame arrives |
| `getTransformMatrix()` | Corrects orientation/scaling |
| `getTimestamp()` | Gets frame capture time |

---

## 📌 SurfaceTexture vs SurfaceView

| Feature | `SurfaceTexture` | `SurfaceView` |
|---|---|---|
| GPU target | GLES texture | Display framebuffer |
| Producer example | Camera, MediaCodec | MediaCodec, OpenGL, etc. |
| Consumer | App via GLES | SurfaceFlinger |
| Use case | OpenGL rendering, filters, 3D | Video playback, Camera preview |
| Thread safe? | Must manage with EGL contexts carefully | Easier for UI rendering |

---

Let me know if you'd like a simple code example showing how to use `SurfaceTexture` with OpenGL or camera!

SurfaceTexture vs SurfaceView explain in simple words

Sure! Let's break down **SurfaceTexture vs SurfaceView** in **very simple words** with examples.

---

## 🔍 What's the difference between SurfaceTexture and SurfaceView?

| Feature | SurfaceTexture | SurfaceView |
|---|---|---|
| 📺 Output target | OpenGL texture (GPU) | Screen (UI window) |
| 🎥 Who uses it? | Camera + OpenGL, video + effects | Video playback, camera preview |
| 🧠 CPU or GPU? | GPU-only rendering | Usually SurfaceFlinger (UI compositor) |
| 🧵 Threading | App controls OpenGL thread | System handles rendering on separate layer |
| 🔁 Repaints | Manual via **`updateTexImage()`** | Automatic by system |
| 🔐 Secure video | Supported (with protected memory) | Also supported |

---

# 🍎 Imagine:

### 🍌 SurfaceView = Window with a video player

- Like playing a YouTube video in an app.
- You **can't easily apply OpenGL effects** to it.
- It's handled by **SurfaceFlinger**, not by your app.

### 🍉 SurfaceTexture = Live video as a texture

- Like a video on a 3D spinning cube in a game.
- You can **draw it with OpenGL**, apply filters, rotate it, warp it, etc.
- App controls when to render, how to render.

---

# 🧠 Easy analogy:

| Thing | Analogy |
|---|---|
| **`SurfaceView`** | A screen playing a movie |
| **`SurfaceTexture`** | A projector showing a movie on any surface (wall, cube, etc.) |
| OpenGL | The one holding the projector |
| **`updateTexImage()`** | Load next movie frame onto wall |

---

# ✅ Use SurfaceTexture when:

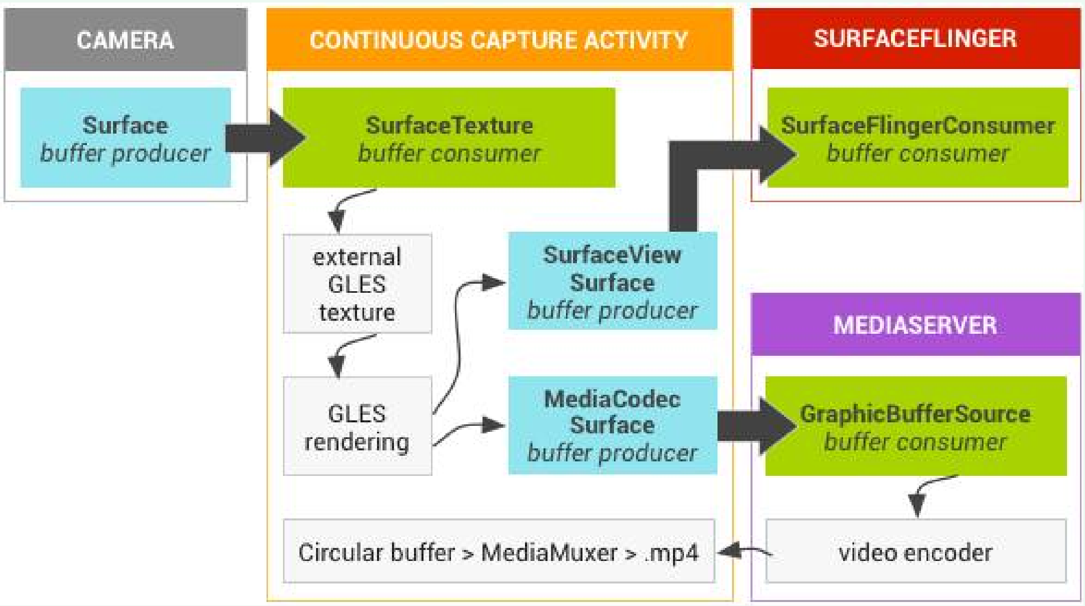
- You want to show video **on a 3D object or with GPU effects**
- You're building a game or AR/VR app
- You need **live camera feed in OpenGL**

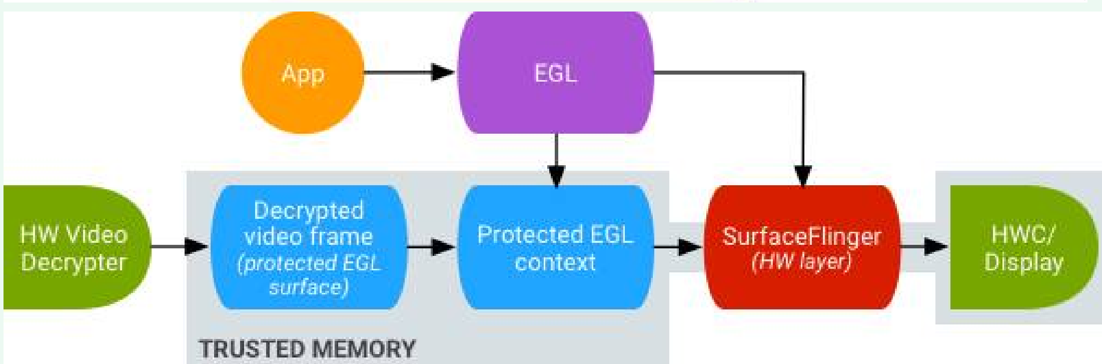
# ✅ Use SurfaceView when:

- You just want to show a video or camera preview **quickly**
- You don't need any OpenGL effects
- You want better performance for **simple video playback**

---

Would you like a simple diagram or code comparison between the two?




Let's break down the difference between `SurfaceTexture` and `SurfaceView` using your diagrams and some simple explanations:

---

## 🎥 What is a `SurfaceTexture`?

- Think of `SurfaceTexture` as a **hidden camera feed receiver**.

- It **receives image frames** (like from a camera or video decoder) and lets you **use those frames as OpenGL textures**.
- It's **not automatically visible on screen**. You decide how and where to show it.

📌 **Main Use:**
Useful when you want to **process camera frames in OpenGL**, like adding effects or combining with 3D content.

📌 **In the first diagram:**

- Camera sends frames to **`SurfaceTexture`** (green).
- You render them in OpenGL.
- You can draw the final result onto another surface, like **`SurfaceView`** or **`MediaCodec`**.

---

## 📺 What is a `SurfaceView`?

- Think of **`SurfaceView`** as a **window for displaying content on screen**.
- It is a **visible drawing area** managed by the system, tied to **hardware rendering**.
- It acts as a **buffer producer**, sending frames to **SurfaceFlinger** to be shown on the screen.

📌 **Main Use:**
Good when you just want to **show video or camera preview directly**, without OpenGL.

📌 **In the first diagram:**

- After **`SurfaceTexture`** or OpenGL rendering, the content can be sent to **`SurfaceView`** to show it on screen.
- **`SurfaceView`** becomes visible via SurfaceFlinger (which handles display composition).

---

## 🔄 Key Differences Summary:

| Feature | SurfaceTexture | SurfaceView |
|---|---|---|
| Visibility | Not visible directly | Directly visible on screen |
| Purpose | OpenGL processing of frames | Displaying content directly |
| Output | OpenGL texture | On-screen surface |
| Common Use Case | Camera frame processing, filtering | Video playback, camera preview |
| Controlled By | App via OpenGL | System compositor (SurfaceFlinger) |
| Buffer Role | **Consumer** (receives frames) | **Producer** (sends frames to SurfaceFlinger) |

---

## 🔐 Second Diagram (Secure Path):

In the secure video playback diagram:

- A **protected EGL context** is used to display **decrypted video** securely.
- The video goes through EGL → SurfaceFlinger → HWC without touching normal app memory.
- This uses **SurfaceTexture or protected surfaces** in EGL to avoid leakage of secure content.

---

## 🧠 Tip to Remember:

> Use **SurfaceTexture** when you need **flexibility and GPU processing**.
> Use **SurfaceView** when you just want to **show content directly and efficiently**.

Let me know if you want an example using camera preview with both approaches.

TextureView

bookmark_border
The TextureView class is a view object that combines a view with a SurfaceTexture.

Rendering with OpenGL ES
A TextureView object wraps a SurfaceTexture, responding to callbacks and acquiring new buffers. When a TextureView acquires new buffers, a TextureView issues a view invalidate request and draws using the contents of the newest buffer as its data source, rendering wherever and however the view state indicates it should.

OpenGL ES (GLES) can render on a TextureView by passing the SurfaceTexture to the EGL creation call, but this creates a problem. When GLES renders on a TextureView, BufferQueue producers and consumers are in the same thread, which can cause the buffer swap call to stall or fail. For example, if a producer submits several buffers in quick succession from the UI thread, the EGL buffer swap call needs to dequeue a buffer from the BufferQueue. However, because the consumer and producer are on the same thread, there won't be any buffers available and the swap call hangs or fails.

To ensure that the buffer swap doesn't stall, BufferQueue always needs a buffer available to be dequeued. To implement this, BufferQueue discards the contents of the previously acquired buffer when a new buffer is queued and places restrictions on minimum and maximum buffer counts to prevent a consumer from consuming all buffers at once.

Choosing SurfaceView or TextureView
Note: In API 24 and higher, it's recommended to implement SurfaceView instead of

TextureView.
SurfaceView and TextureView fill similar roles and are both citizens of the view hierarchy. However, SurfaceView and TextureView have different implementations. A SurfaceView takes the same parameters as other views, but SurfaceView contents are transparent when rendered.

A TextureView has better alpha and rotation handling than a SurfaceView, but a SurfaceView has performance advantages when compositing UI elements layered over videos. When a client renders with a SurfaceView, the SurfaceView provides the client with a separate composition layer. SurfaceFlinger composes the separate layer as a hardware overlay if supported by the device. When a client renders with a TextureView, the UI toolkit composites the TextureView's content into the view hierarchy with the GPU. Updates to the content may cause other view elements to redraw, for example, if the other views are positioned on top of a TextureView. After view rendering completes, SurfaceFlinger composites the app UI layer and all other layers, so that every visible pixel is composited twice.

Note: DRM-protected video can be presented only on an overlay plane. Video players that support protected content must be implemented with SurfaceView.
Case Study: Grafika's Play Video
Grafika's Play Video includes a pair of video players, one implemented with TextureView and one implemented with SurfaceView. The video decoding portion of the activity sends frames from MediaCodec to a surface for both TextureView and SurfaceView. The biggest difference between the implementations are the steps required to present the correct aspect ratio.

Scaling SurfaceView requires a custom implementation of FrameLayout. WindowManager needs to send a new window position and new size values to SurfaceFlinger. Scaling a TextureView's SurfaceTexture requires configuring a transformation matrix with TextureView#setTransform().

After presenting the correct aspect ratio, both implementations follow the same pattern. When SurfaceView/TextureView creates the surface, the app code enables playback. When a user taps play, it starts a video decoding thread, with the surface as the output target. After that, the app code doesn't do anything—composition and display are handled by SurfaceFlinger (for the SurfaceView) or by the TextureView.

Case Study: Grafika's Double Decode
Grafika's Double Decode demonstrates manipulation of the SurfaceTexture inside a TextureView.

Grafika's Double Decode uses a pair of TextureView objects to show two videos playing side by side, simulating a video conferencing app. When the orientation of the screen changes and the activity restarts, the MediaCodec decoders don't stop, simulating playback of a real-time video stream. To improve efficiency, the client should keep the surface alive. The surface is a handle to the producer interface in the SurfaceTexture's BufferQueue. Because the TextureView manages the SurfaceTexture, the client needs to keep the SurfaceTexture alive to keep the surface alive.

To keep the SurfaceTexture alive, Grafika's Double Decode obtains references to

SurfaceTextures from the TextureView objects and saves them in a static field. Then, Grafika's Double Decode returns false from TextureView.SurfaceTextureListener#onSurfaceTextureDestroyed() to prevent the destruction of the SurfaceTexture. TextureView then passes a SurfaceTexture to onSurfaceTextureDestroyed() that can be maintained across the activity configuration change, which the client passes to the new TextureView through setSurfaceTexture().

Separate threads drive each video decoder. Mediaserver sends buffers with decoded output to the SurfaceTextures, the BufferQueue consumers. The TextureView objects perform rendering and execute on the UI thread.

Implementing Grafika's Double Decode with SurfaceView is harder than implementing with TextureView because SurfaceView objects destroy surfaces during orientation changes. Additionally, using SurfaceView objects adds two layers, which isn't ideal because of the limitations on the number of overlays available on the hardware.

Here's a **simple explanation** of `TextureView` and how it compares to `SurfaceView` and `SurfaceTexture`, with the key ideas from your notes:

---

## 📦 What is a `TextureView`?

- `TextureView` is a **View** that wraps a **SurfaceTexture**.
- You can treat it like a **normal UI widget**, but it internally gives you access to the image buffer (`SurfaceTexture`) so you can do:
  - Camera preview
  - Video rendering
  - OpenGL effects

---

## 🔁 How `TextureView` Works

- It uses **SurfaceTexture** internally.
- It listens for **new frames** and then **draws them like any other view** (e.g., in a layout).
- If used with OpenGL ES (GLES), it can render frames to its internal texture.

🔧 **But be careful!** If producer and consumer (e.g., camera and GL) are on the same thread, it can stall due to buffer starvation.

---

## 🔍 TextureView vs SurfaceView — Simple Comparison

| Feature | TextureView | SurfaceView |
|---|---|---|
| Part of view hierarchy | ✅ Yes | ✅ Yes (but uses a separate window) |
| Compositing method | **GPU-based** (drawn with UI views) | **Hardware overlay** (separate layer) |
| Alpha / Rotation support | ✅ Good (can easily scale/rotate) | ❌ Limited |
| Performance (video playback) | ❌ Slightly worse (more GPU work) | ✅ Better (offloaded to hardware) |
| DRM / Protected video | ❌ Not supported | ✅ Required for DRM video |
| Surface lifespan | 🔁 Can **persist across config changes** | ❌ Destroyed on config change |
| Use OpenGL directly? | ✅ Yes (via SurfaceTexture) | ✅ Yes (via Surface) |

---

## 📸 Simple Use Cases

| Use Case | Recommended View |
|---|---|
| Camera preview with GL effects | `TextureView` |
| Basic video playback | `SurfaceView` |
| Display DRM-protected content | `SurfaceView` |
| Real-time 2D/3D UI composition | `TextureView` |
| Efficient fullscreen playback | `SurfaceView` |

---

## 🧠 Important Notes

- In **API 24+ (Android 7.0+)**, **SurfaceView** is preferred for performance.
- `TextureView` is better when:
  - You want smooth alpha blending
  - You need rotation and scaling
  - You want to easily keep the surface alive across configuration changes
- `SurfaceView` is better when:
  - You care about **battery life and performance**
  - You need **DRM protection**
  - You want to offload rendering to hardware overlays

---

Let me know if you want a diagram or example code comparing them visually!

SurfaceFlinger and WindowManager

bookmark_border
SurfaceFlinger accepts buffers, composes buffers, and sends buffers to the display. WindowManager provides SurfaceFlinger with buffers and window metadata, which SurfaceFlinger uses to composite surfaces to the display.

SurfaceFlinger
SurfaceFlinger can accept buffers in two ways: through BufferQueue and SurfaceControl, or through ASurfaceControl.

One way SurfaceFlinger accepts buffers is through BufferQueue and SurfaceControl. When an app comes to the foreground, it requests buffers from WindowManager. WindowManager then requests a layer from SurfaceFlinger. A layer is a combination of a surface, which contains the BufferQueue, and a SurfaceControl instance, which contains the layer metadata like the display frame. SurfaceFlinger creates the layer and sends it to WindowManager. WindowManager then sends the surface to the app, but keeps the SurfaceControl instance to manipulate the appearance of the app on the screen.

Android 10 adds ASurfaceControl, which is another way that SurfaceFlinger can accept buffers. ASurfaceControl combines a surface and a SurfaceControl instance into one transaction package that is sent to SurfaceFlinger. ASurfaceControl is associated with a layer, which apps update through ASurfaceTransaction instances. Apps then get information about ASurfaceTransaction instances through callbacks that pass ASurfaceTransactionStats containing information, such as latch time, acquire times, and so on.

The following table includes more details about ASurfaceControl and its associated components:

Component	Description
ASurfaceControl	Wraps SurfaceControl and enables an app to create SurfaceControl instances that correspond to layers on the display.

Can be created as a child of ANativeWindow or as a child of another ASurfaceControl instance.
ASurfaceTransaction	Wraps Transaction to enable the client to edit a layer's descriptive properties, such as geometry, and sends the updated buffers to SurfaceFlinger.
ASurfaceTransactionStats	Sends information about transactions that have been presented, such as latch time, acquire times, and previous release fence, to an app through a preregistered callback.
Though apps can submit buffers at any time, SurfaceFlinger only wakes up to accept buffers between display refreshes, which can differ depending on the device. This minimizes memory usage and avoids visible tearing on the screen, which can occur when updating the display mid-refresh.

When the display is between refreshes, the display sends the VSync signal to SurfaceFlinger.

The VSync signal indicates that the display can be refreshed without tearing. When SurfaceFlinger receives the VSync signal, SurfaceFlinger walks through its list of layers looking for new buffers. If SurfaceFlinger finds a new buffer, SurfaceFlinger acquires the buffer; if not, SurfaceFlinger continues to use the previously acquired buffer. SurfaceFlinger must always display something, so it hangs on to one buffer. If no buffers have ever been submitted on a layer, the layer is ignored.

After SurfaceFlinger has collected all buffers for visible layers, it asks the Hardware Composer (HWC) how composition should be performed. If the HWC marks layer composition type as client composition, SurfaceFlinger composites those layers. Then, SurfaceFlinger passes the output buffer to the HWC.

WindowManager
WindowManager controls Window objects, which are containers for View objects. Window objects are always backed by Surface objects. WindowManager oversees lifecycles, input and focus events, screen orientation, transitions, animations, position, transforms, z-order, and many other aspects of a window. WindowManager sends all of the window metadata to SurfaceFlinger so SurfaceFlinger can use that data to composite surfaces on the display.

Pre-rotation
Many hardware overlays don't support rotation (and even if they do, it costs processing power); the solution is to transform the buffer before it reaches SurfaceFlinger. Android supports a query hint (NATIVE_WINDOW_TRANSFORM_HINT) in ANativeWindow to represent the most likely transform to be applied to the buffer by SurfaceFlinger. GL drivers can use this hint to pre-transform the buffer before it reaches SurfaceFlinger so that when the buffer arrives, it's correctly transformed.

For example, when receiving a hint to rotate 90 degrees, generate and apply a matrix to the buffer to prevent it from running off the end of the page. To save power, do this pre-rotation. For details, see the ANativeWindow interface defined in system/core/include/system/window.h.

Here's a **simple breakdown** of how **SurfaceFlinger** and **WindowManager** work together in Android graphics:

---

## 🧱 1. What is SurfaceFlinger?

Think of **SurfaceFlinger** as the **"screen painter"**. Its job is:

- 📥 Accept **buffers** (images) from apps.
- 🧩 **Compose** all visible layers (windows, UI, video, etc.) together.
- 📺 Send the final composed image to the **display**.

### 🔄 When does it work?

- SurfaceFlinger only runs when the **display is ready to refresh**, triggered by **VSync** (vertical sync) to avoid visual glitches like tearing.

---

## 🧰 2. What is WindowManager?

Think of **WindowManager** as the **"window organizer"**. It manages:

- Which **windows** are visible.
- Where each window is located on screen.
- Which window has **focus**, handles **input**, or changes size.
- Animations, transitions, screen rotations.

🔁 It tells SurfaceFlinger:

- “Here’s what to draw”
- “Where to draw it”
- “How to layer it (z-order)”

---

## 🔄 How They Work Together

1. ✋ App asks **WindowManager** for a window (usually via **`Activity`** or **`Dialog`**).
2. 🪟 WindowManager requests a **layer** from SurfaceFlinger.
    - A **layer** = **`Surface`** (buffer queue) + **`SurfaceControl`** (metadata).
3. 🎨 SurfaceFlinger sends back a surface to the app, but keeps the SurfaceControl to manage rendering.
4. 🖌️ App draws onto the Surface via OpenGL / Vulkan / Canvas.
5. 🧠 On every **VSync**, SurfaceFlinger:
    - Collects new buffers.
    - Decides how to **compose** them (client composition or hardware overlay).
    - Passes the final frame to the **Hardware Composer (HWC)** for display.

---

## 🎛️ Newer API: ASurfaceControl (Android 10+)

Instead of separate Surface + SurfaceControl, **`ASurfaceControl`** combines both:

| Component | What it does |
|---|---|
| **ASurfaceControl** | Manages layer (like SurfaceControl) + surface |

| Component | What it does |
|---|---|
| **ASurfaceTransaction** | Groups buffer + metadata updates together |
| **ASurfaceTransactionStats** | Gives feedback on when/how buffers were used |

→ Helps with **atomic updates**, **animation timing**, and **latency info**.

---

# 🔄 What is Pre-Rotation?

🔄 **Problem**: Some devices can't rotate hardware overlay layers (like video).

🛠️ **Solution**:

- Before drawing, the system gives a **"hint"** to apps (`NATIVE_WINDOW_TRANSFORM_HINT`).
- App pre-rotates the buffer (e.g., rotates camera preview).
- Saves GPU/CPU work during final composition.

---

# 🧠 Summary Diagram (Simplified)

```
plaintext

App       →    WindowManager         →     SurfaceFlinger     →    Display
             (Manages window info)        (Composes layers)        (Shows final image)
 ↓                          ↑
Surface ← SurfaceControl ←+
(Buffer Queue)     (Layer metadata)
```

Let me know if you'd like a visual flowchart or to dive deeper into **VSync**, **HWC**, or **transactions**!

how app, window manager, surfaceflinger, and others communicate each explain in simple words

Here's a **simple explanation** of how Android components like **App**, **WindowManager**, **SurfaceFlinger**, and **Hardware Composer (HWC)** talk to each other — like a team working together to show your app on the screen:

---

# 🧠 Overall Analogy:

Imagine you're making a movie 🎬:

- **App** is the **Director** 🎥 (it decides what to show).

- **WindowManager** is the **Producer** 📋 (organizes scenes, who goes first).
- **SurfaceFlinger** is the **Editor** 🎞️ (mixes all scenes into one frame).
- **Hardware Composer (HWC)** is the **Projector Operator** 🎦 (shows it on screen).

---

## 📱 1. App → WindowManager

### Communication:

✅ **Binder IPC**

### What is passed?

- App asks to **create a window** (via Activity/View/SurfaceView).
- WindowManager gives it a `Surface` to draw on (a **buffer queue**).

---

## 🧰 2. WindowManager → SurfaceFlinger

### Communication:

✅ **Binder IPC** (uses `SurfaceControl` under the hood)

### What is passed?

- Tells SurfaceFlinger:
  - Where each window goes on screen.
  - Their order (front or back).
  - Size, rotation, transparency.

---

## 🎨 3. App → SurfaceFlinger

### Communication:

✅ **Shared memory (via BufferQueue)**

### What is passed?

- App renders frames into a **Surface**, which is a buffer queue.
- These buffers are passed **directly** to SurfaceFlinger.

💡 No binder here — this is **fast zero-copy** using shared memory via **gralloc**.

# 🧠 4. SurfaceFlinger → HWC (Hardware Composer)

### Communication:

✅ **HIDL / AIDL / HAL APIs**

### What is passed?

- A **display list**:
  - Which buffers to show.
  - Where and how to show them (position, transform, layer).
- HWC tries to do **hardware overlays** (zero GPU) if possible.

---

# 🌀 5. VSync triggers redraw

- **Display** sends VSync signal (≈ 60 times per second).
- **Choreographer** tells App to draw.
- SurfaceFlinger also wakes up and composes new frame.

---

# 💬 Summary Table

| Component | Talks To | How | What They Share |
|---|---|---|---|
| **App** | WindowManager | **Binder** | Requests window, gets Surface |
| **App** | SurfaceFlinger | **BufferQueue (shared mem)** | Renders frames via Surface |
| **WindowManager** | SurfaceFlinger | **SurfaceControl + Binder** | Z-order, size, visibility |
| **SurfaceFlinger** | HWC (HAL) | **HAL / AIDL / HIDL** | Layer composition, display commands |
| **Display** | SurfaceFlinger | **VSync signal** | When to draw next frame |

---

# 🔁 Flow Example

1. App starts → asks WindowManager for a window.
2. WindowManager gives `Surface` (buffer queue).
3. App draws into the Surface.
4. SurfaceFlinger reads from Surface, composes full frame.
5. HWC displays the composed image.

6. Repeat on next **VSync**.

---

Let me know if you'd like this shown in a **visual flowchart** or want to explore one part deeper (e.g., buffer queue, SurfaceControl, or Binder IPC)!

↳
📱 1. App → WindowManager ------------------------ ### Communication: ✅ **Binder IPC** ### What is passed? * App asks to **create a window** (via Activity/View/SurfaceView). * WindowManager gives it a `Surface` to draw on (a **buffer queue**).
explain in details

Let's explain in **simple and detailed terms** how **App**, **WindowManager**, **SurfaceFlinger**, and other components communicate in the Android graphics system:

---

## 🧱 Basic Concept: Who Are the Players?

| Component | Role |
|---|---|
| **App (Activity/View)** | Draws UI (buttons, images, etc.) |
| **WindowManager** | Manages screen layout, windows, z-order |
| **SurfaceFlinger** | Compositor: takes images from apps and shows on screen |
| **Surface** | A handle to draw into (wraps a buffer queue) |
| **BufferQueue** | The queue between producer and consumer of frames |

---

## 📱 1. App → WindowManager

### Communication:

- **Binder IPC**

### What's happening?

- When your app starts (or shows a new **`Activity`**), it asks the **WindowManager** to create a window.
- It may contain a **`View`**, **`SurfaceView`**, or **`SurfaceTexture`**.

### What is returned?

- The **WindowManager** responds with a **`Surface`** — this is a **producer** side of a **BufferQueue**.

- The app can now draw to this `Surface`.

---

## 🖼️ 2. App uses Surface to draw UI

### How?

- Your app (or the UI toolkit) uses **OpenGL**, **Canvas**, or **HardwareRenderer** to draw.
- This drawing fills up **GraphicBuffers** and pushes them into the **BufferQueue**.

---

## 🪟 3. SurfaceView (or SurfaceTexture) connects to SurfaceFlinger

- The **consumer** side of the **BufferQueue** is often handled by **SurfaceFlinger**.
- It reads the latest buffers from each app, and composes the final screen.

---

## 🧠 4. WindowManager → SurfaceFlinger

### Communication:

- Also through **Binder IPC**

### What?

- When WindowManager wants to show something (e.g., a new app or window), it tells **SurfaceFlinger**:
  - "Here's a new Layer."
  - "Put this app's Surface here (x, y), with these bounds."

---

## 🧩 5. SurfaceFlinger → Hardware Composer (HWC)

- SurfaceFlinger **composites** all layers.
- Then it hands them off to the **Hardware Composer HAL**, which interacts with the **display hardware (GPU/DSI panel)**.

---

## 🔁 Summary: Flow of communication

```pgsql
App ----> WindowManager (Binder IPC)
   (requests window/surface)

WindowManager ----> Surface (creates buffer queue)
App draws to Surface (via OpenGL, Canvas, etc.)

Surface ----> BufferQueue (producer side)
SurfaceFlinger ----> BufferQueue (consumer side)
   (reads buffers from apps)

SurfaceFlinger composites → passes to HWC → screen
```

---

## 📦 BufferQueue Roles:

| Component | Producer | Consumer |
|---|---|---|
| **App** | ✅ | ❌ |
| **SurfaceFlinger** | ❌ | ✅ |
| **SurfaceView** | ✅ | ✅ (draw + pass) |
| **Camera + SurfaceTexture** | ✅ (camera) | ✅ (GPU) |

---

Would you like this visualized in a block diagram next?

System properties in SurfaceFlinger

bookmark_border
The ConfigStore HAL remains in AOSP to support legacy vendor partitions. On devices running Android 10+, surfaceflinger reads system properties first; if no system property is defined for a config item in SurfaceFlingerProperties.sysprop, surfaceflinger falls back to the ConfigStore HAL.

Build flags and system properties
Each build flag in ConfigStore has a matching system property, as shown in the following table. For details on these properties, refer to
frameworks/native/services/surfaceflinger/sysprop/SurfaceFlingerProperties.sysprop.

Build flags	System properties
TARGET_FORCE_HWC_FOR_VIRTUAL_DISPLAYS
ro.surface_flinger.force_hwc_copy_for_virtual_displays
TARGET_HAS_HDR_DISPLAY	ro.surface_flinger.has_HDR_display
TARGET_HAS_WIDE_COLOR_DISPLAY	ro.surface_flinger.has_wide_color_display
NUM_FRAMEBUFFER_SURFACE_BUFFERS

ro.surface_flinger.max_frame_buffer_acquired_buffers
MAX_VIRTUAL_DISPLAY_DIMENSION ro.surface_flinger.max_virtual_display_dimension
PRIMARY_DISPLAY_ORIENTATION ro.surface_flinger.primary_display_orientation
PRESENT_TIME_OFFSET_FROM_VSYNC_NS
ro.surface_flinger.present_time_offset_from_vsync_ns
TARGET_RUNNING_WITHOUT_SYNC_FRAMEWORK
ro.surface_flinger.running_without_sync_framework
SF_START_GRAPHICS_ALLOCATOR_SERVICE ro.surface_flinger.start_graphics_allocator_service
TARGET_USE_CONTEXT_PRIORITY ro.surface_flinger.use_context_priority
USE_VR_FLINGER ro.surface_flinger.use_vr_flinger
VSYNC_EVENT_PHASE_OFFSET_NS ro.surface_flinger.vsync_event_phase_offset_ns
SF_VSYNC_EVENT_PHASE_OFFSET_NS ro.surface_flinger.vsync_sf_event_phase_offset_ns
Using SurfaceFlingerProperties
The following examples show how to use the SurfaceFlingerProperties library and the Surface Flinger property ro.surface_flinger.vsync_event_phase_offset_ns.

To query for the value of the property in reference, use the property's api_name as the function name.

Include SurfaceFlingerProperties in your build file, as follows:

```
cc_binary {
   name: "cc_client",
   srcs: ["baz.cpp"],
   shared_libs: ["SurfaceFlingerProperties"],
}
java_library {
   name: "JavaClient",
   srcs: ["foo/bar.java"],
   libs: ["SurfaceFlingerProperties"],
}
```

The following Java code snippet uses the ro.surface_flinger.vsync_event_phase_offset_ns system property:

```
import android.sysprop.SurfaceFlingerProperties;
...

static void foo() {
   ...
   boolean temp = SurfaceFlingerProperties.vsync_event_phase_offset_ns().orElse(true);
   ...
}
...
```

The following C++ code snippet uses the ro.surface_flinger.vsync_event_phase_offset_ns system

```
property:


#include <SurfaceFlingerProperties.sysprop.h>
using namespace android::sysprop;

...

void bar() {
   ...
   bool temp = SurfaceFlingerProperties::vsync_event_phase_offset_ns(true);
   ...
}
...
```

Here's a **simple and detailed explanation** of how **system properties** work in **SurfaceFlinger**, and what role they play.

---

## 🔧 What are "System Properties"?

System properties in Android are like **global settings**—key-value pairs that can be **read by any part of the system** (apps, services, HALs). They are stored in memory and often set at boot time or via AOSP configs.

Example:

```ini
ro.surface_flinger.use_context_priority = true
```

---

## 🧠 What is SurfaceFlinger?

**SurfaceFlinger** is the **Android system compositor**:

- It gets graphic buffers from apps
- Composites them into a single image
- Sends the final image to the display (via HWC or GPU)

To control SurfaceFlinger's behavior (e.g. vsync, HDR), Android provides **system properties** and build-time flags.

---

# 📜 How System Properties Work in SurfaceFlinger

## ✅ Priority:

1. SurfaceFlinger **first checks system properties** (like `ro.surface_flinger.use_context_priority`)
2. If a system property **is not set**, it **falls back to old ConfigStore HAL** (for legacy support)

---

# 🔄 Mapping: Build flags ↔ System properties

| Build-time flag (in device.mk) | System Property (runtime) |
|---|---|
| `TARGET_HAS_HDR_DISPLAY` | `ro.surface_flinger.has_HDR_display` |
| `NUM_FRAMEBUFFER_SURFACE_BUFFERS` | `ro.surface_flinger.max_frame_buffer_acquired_buffers` |
| `PRESENT_TIME_OFFSET_FROM_VSYNC_NS` | `ro.surface_flinger.present_time_offset_from_vsync_ns` |
| ... | ... |

So, a **flag like**:

```make
TARGET_HAS_HDR_DISPLAY := true
```

Can be overridden at runtime by setting:

```bash
adb shell setprop ro.surface_flinger.has_HDR_display true
```

---

# 📦 How to Use These in Code?

## ✅ C++ usage

```cpp
#include <SurfaceFlingerProperties.sysprop.h>
using namespace android::sysprop;

void bar() {
    bool temp = SurfaceFlingerProperties::vsync_event_phase_offset_ns(true); // default: true
}
```

## ✅ Java usage

```java
import android.sysprop.SurfaceFlingerProperties;

void foo() {
    boolean temp = SurfaceFlingerProperties.vsync_event_phase_offset_ns().orElse(true);
}
```

## 📚 Typical Use Cases for SurfaceFlinger Properties

| Property Name | Use |
|---|---|
| `ro.surface_flinger.use_context_priority` | GPU context scheduling priority |
| `ro.surface_flinger.has_HDR_display` | Enables HDR composition |
| `ro.surface_flinger.primary_display_orientation` | Portrait/Landscape handling |
| `ro.surface_flinger.vsync_event_phase_offset_ns` | Controls phase offset for smooth rendering |
| `ro.surface_flinger.running_without_sync_framework` | For non-vsync/non-choreographer display |

## ⚠️ Notes:

- These properties are typically **read-only (`ro.`)** and **set at build time** or via boot args.
- Changing them at runtime may not take effect until **reboot** or **restart of SurfaceFlinger**.

## 🏁 Summary

| Concept | Meaning |
|---|---|
| System Property | Runtime config used by components like SurfaceFlinger |
| Build Flag | Compile-time value that becomes a system property |
| Fallback | If system property isn't set, fallback to legacy ConfigStore HAL |
| Usage | Access via **`SurfaceFlingerProperties`** in C++ or Java |

Let me know if you want a breakdown of any specific property or how these affect performance (e.g. triple buffering, vsync tuning).

confusing

No problem — let me explain **SurfaceFlinger system properties** in **simple, clear steps**:

## 🔷 What are system properties in Android?

System properties are like **key-value settings** used by Android components to control behavior.
For example:

```bash
ro.surface_flinger.use_vr_flinger = true
```

This tells SurfaceFlinger: *"Use VR Flinger mode."*

---

## 🔷 What is SurfaceFlinger?

SurfaceFlinger is the **display compositor** in Android.
It takes graphics buffers from apps (via Surface/SurfaceView) and **composes** them into one final image to send to the screen.

---

## 🔷 Why does SurfaceFlinger use system properties?

To control:

- Display behavior
- Performance tuning
- Hardware features (HDR, wide color, etc.)
- VSync timing
- Graphics allocator startup

---

## 🔷 Where are these system properties listed?

They are defined in:

```bash
frameworks/native/services/surfaceflinger/sysprop/SurfaceFlingerProperties.sysprop
```

Each **build-time flag** has a **matching system property**, like this:

| Build Flag | System Property |
| --- | --- |
| `TARGET_FORCE_HWC_FOR_VIRTUAL_DISPLAYS` | `ro.surface_flinger.force_hwc_copy_for_virtual_display` |
| `TARGET_HAS_HDR_DISPLAY` | `ro.surface_flinger.has_HDR_display` |
| `TARGET_HAS_WIDE_COLOR_DISPLAY` | `ro.surface_flinger.has_wide_color_display` |
| `NUM_FRAMEBUFFER_SURFACE_BUFFERS` | `ro.surface_flinger.max_frame_buffer_acquired_buffers` |

| Build Flag | System Property |
|---|---|
| `VSYNC_EVENT_PHASE_OFFSET_NS` | `ro.surface_flinger.vsync_event_phase_offset_ns` |

## 🔷 How to use these properties in code?

### ✅ In Java:

```java
import android.sysprop.SurfaceFlingerProperties;

boolean value = SurfaceFlingerProperties.vsync_event_phase_offset_ns().orElse(true);
```

### ✅ In C++:

```cpp
#include <SurfaceFlingerProperties.sysprop.h>
using namespace android::sysprop;

bool value = SurfaceFlingerProperties::vsync_event_phase_offset_ns(true);
```

This will check the value of `ro.surface_flinger.vsync_event_phase_offset_ns`.

## 🔷 What happens if the property is not set?

If the system property is **not defined**, SurfaceFlinger will fall back to **ConfigStore HAL**, which is the older mechanism.

## 🔷 Real-world Example

Let's say your hardware supports HDR.

You set:

```bash
ro.surface_flinger.has_HDR_display = true
```

Now SurfaceFlinger enables HDR rendering path when composing frames.

## 🧠 Summary

| Term | Meaning |
|---|---|
| System Property | Runtime setting used by Android services |
| SurfaceFlinger | The service that draws everything on screen |
| SurfaceFlingerProperties | A code interface to read those system properties in Java/C++ |
| ConfigStore HAL | Legacy way to get hardware config (used if no system property is set) |

---

Let me know if you want a diagram or example using one specific property (e.g., VSync offset).

Hardware Composer HAL

bookmark_border
The Hardware Composer (HWC) HAL determines the most efficient way to composite buffers with the available hardware. As a HAL, its implementation is device-specific and usually done by the display hardware OEM.

The value of this approach is easy to recognize when you consider overlay planes, which composite multiple buffers in the display hardware rather than the GPU. For example, consider a typical Android phone in portrait orientation, with the status bar on top, navigation bar at the bottom, and app content everywhere else. The contents for each layer are in separate buffers. You can handle composition using either of the following methods:

Rendering the app content into a scratch buffer, then rendering the status bar over it, the navigation bar on top of that, and finally passing the scratch buffer to the display hardware.
Passing all three buffers to the display hardware and instructing it to read data from different buffers for different parts of the screen.
The latter approach can be significantly more efficient.

Display processor capabilities vary significantly. The number of overlays, whether layers can be rotated or blended, and restrictions on positioning and overlap can be difficult to express through an API. To accommodate these options, the HWC performs following calculations:

SurfaceFlinger provides HWC with a full list of layers and asks, "How do you want to handle this?"
HWC responds by marking each layer as device or client composition.
SurfaceFlinger takes care of any client, passing the output buffer to HWC, and lets HWC handle the rest.
Because hardware vendors can custom tailor decision-making code, it's possible to get the best performance out of every device.

Overlay planes may be less efficient than GL composition when nothing on the screen is changing. This is particularly true when overlay contents have transparent pixels and overlapping layers are blended. In such cases, the HWC can request GLES composition for some or all layers and retain the composited buffer. If SurfaceFlinger asks to composite the same set of buffers, the HWC can show the previously composited scratch buffer. This can improve the battery life of an idle device.

Android devices typically support four overlay planes. Attempting to composite more layers than overlays causes the system to use GLES composition for some of them, meaning the number of layers used by an app can have a measurable impact on power consumption and performance.

Let's break this down step by step and **explain the role of the Hardware Composer (HWC) HAL** in a **simple and detailed way**, along with how it works with **SurfaceFlinger**, the **GPU**, and **display hardware**.

---

## 🔧 What is the Hardware Composer HAL (HWC)?

**HWC** is a **Hardware Abstraction Layer** in Android that knows how to **efficiently draw things on the screen using the display hardware**.

- It's written by the **display hardware vendor** (Qualcomm, MediaTek, etc.).
- It tells the system:
  "Here's the best way to show these things on screen without wasting battery or GPU."

---

## 🧱 Scenario: What are we trying to draw?

Imagine your phone screen:

- A game or app content (full screen)
- A status bar on top
- A navigation bar at the bottom

Each of these (app, status bar, nav bar) is a **layer**, and each layer has its own **buffer**.

---

## 🧠 How can these be drawn?

### 🅰️ Option 1: Using the GPU (GL Composition)

- GPU draws **app** → writes to **scratch buffer**
- GPU draws **status bar** on top of scratch buffer
- GPU draws **nav bar** next

- Result: One final composed image sent to display

✅ Works on all devices
❌ GPU uses power and time (especially if nothing changes)

---

### 🅱️ Option 2: Using Hardware Composer (Overlay Planes)

- Hardware display controller can **directly combine multiple layers** (like sandwiching transparent slides):
  - “Use buffer A for top part of screen”
  - “Use buffer B for bottom part”
  - “Use buffer C for middle”

✅ More efficient, **saves GPU and battery**
✅ Can skip GPU completely if possible
❌ Limited by **hardware** (only a few overlays, restrictions on blend/rotation)

---

## 🧩 How does the decision happen?

### Step-by-step flow:

1. **SurfaceFlinger** collects all layers to be drawn (app, nav bar, etc.)
2. SurfaceFlinger calls **HWC**:
   “Hey, here are all the layers — can you handle them with the display hardware?”
3. **HWC responds**:
   - Marks each layer:
     - ✅ **Device Composition** (HWC can handle directly)
     - ❌ **Client Composition** (GPU should handle)
4. For **Client Composition**, SurfaceFlinger:
   - Uses **GLES** to combine them into one final buffer
   - Gives that final buffer to **HWC**
5. **HWC then sends the result to the display** (using overlays, or just one composed image)

---

## 📦 Bonus: Reuse and Power Optimization

- If nothing changes on screen, **HWC can reuse a previously composed buffer**.
- This avoids waking up the GPU again → **saves power**, especially when the screen is idle.

## ⚙️ Hardware Limitation

- Most Android devices support only **~4 overlay planes**.
- If there are **too many layers**, HWC will **fall back to GPU** for some layers.
- So: More layers → more GPU → more power used.

## 🔁 Summary (Simple Table)

| Component | Role |
|---|---|
| **SurfaceFlinger** | Collects layers and decides who draws them (GPU or HWC) |
| **GPU (GLES)** | Composes layers in software when HWC can't |
| **HWC HAL** | Offloads layer composition to hardware (if possible) |
| **Display Engine** | Actually displays the final image on screen |

Let me know if you want a diagram to explain this flow or how to check what composition path (HWC or GPU) is used on your device.

Implement Hardware Composer HAL

bookmark_border
The Hardware Composer (HWC) HAL composites layers received from SurfaceFlinger, reducing the amount of composition OpenGL ES (GLES) and the GPU perform.

The HWC abstracts objects, such as overlays and 2D blitters, to composite surfaces and communicates with specialized window composition hardware to composite windows. Use the HWC to composite windows instead of having SurfaceFlinger composite with the GPU. Most GPUs aren't optimized for composition, and when the GPU composes layers from SurfaceFlinger, apps can't use the GPU for their own rendering.

HWC implementations should support:

At least four overlays:
Status bar
System bar
App
Wallpaper/background
Layers that are larger than the display (for example, a wallpaper)
Simultaneous premultiplied per-pixel alpha blending and per-plane alpha blending

Hardware path for protected video playback
RGBA packing order, YUV formats, and tiling, swizzling, and stride properties
To implement the HWC:

Implement a nonoperational HWC and send all composition work to GLES.
Implement an algorithm to delegate composition to the HWC incrementally. For example, delegate only the first three or four surfaces to the overlay hardware of the HWC.
Optimize the HWC. This may include:
Selecting surfaces that maximize the load taken off the GPU and sending them to the HWC.
Detecting whether the screen is updating. If it isn't, delegate composition to GLES instead of the HWC to save power. When the screen updates again, continue to offload composition to the HWC.
Preparing for common use cases such as:
The home screen, which includes the status bar, system bar, app window, and live wallpapers
Full-screen games in portrait and landscape mode
Full-screen video with closed captioning and playback control
Protected video playback
Split-screen multiwindow
Note: Use cases should address regular, predictable cases instead of edge cases to maximize optimization benefits.
HWC primitives
The HWC provides two primitives, layers and displays, to represent composition work and its interaction with the display hardware. The HWC also provides control over VSync and a callback to SurfaceFlinger to notify it when a VSync event occurs.

HIDL interface
Android 8.0 and higher uses a HIDL interface called Composer HAL for binderized IPC between the HWC and SurfaceFlinger. The Composer HAL replaces the legacy hwcomposer2.h interface. If vendors provide a Composer HAL implementation of the HWC, Composer HAL directly accepts HIDL calls from SurfaceFlinger. If vendors provide a legacy implementation of the HWC, Composer HAL loads function pointers from hwcomposer2.h, forwarding HIDL calls into function pointer calls.

The HWC provides functions to determine the properties of a given display; to switch between different display configurations (such as 4k or 1080p resolution) and color modes (such as native color or true sRGB); and to turn the display on, off, or into a low-power mode if supported.

Function pointers
If vendors implement Composer HAL directly, SurfaceFlinger calls its functions through HIDL IPC. For example, to create a layer, SurfaceFlinger calls createLayer() on the Composer HAL.

If vendors implement the hwcomposer2.h interface, Composer HAL calls into hwcomposer2.h function pointers. In hwcomposer2.h comments, HWC interface functions are referred to by lowerCamelCase names that don't exist in the interface as named fields. Almost every function is loaded by requesting a function pointer using getFunction provided by hwc2_device_t. For example, the function createLayer is a function pointer of type HWC2_PFN_CREATE_LAYER, which is returned when the enumerated value HWC2_FUNCTION_CREATE_LAYER is passed into getFunction.

For detailed documentation on Composer HAL functions and HWC function passthrough functions, see composer. For detailed documentation on HWC function pointers, see the hwcomposer2.h.

Layer and display handles
Layers and displays are manipulated by handles generated by the HWC. The handles are opaque to SurfaceFlinger.

When SurfaceFlinger creates a new layer, it calls createLayer, which returns of type Layer for direct implementations or hwc2_layer_t for passthrough implementations. When SurfaceFlinger modifies a property of that layer, SurfaceFlinger passes the hwc2_layer_t value into the appropriate modification function along with any other information needed to make the modification. The hwc2_layer_t type is large enough to hold either a pointer or an index.

Physical displays are created by being hotplugged. When a physical display is hotplugged, the HWC creates a handle and passes the handle to SurfaceFlinger through the hotplug callback. Virtual displays are created by SurfaceFlinger calling createVirtualDisplay() to request a display. If the HWC supports virtual display composition, it returns a handle. Then, SurfaceFlinger delegates the displays's composition to the HWC. If the HWC doesn't support virtual display composition, SurfaceFlinger creates the handle and composites the display.

Display composition operations
Once per VSync, SurfaceFlinger wakes if it has new content to composite. This new content can be new image buffers from apps or a change in the properties of one or more layers. When SurfaceFlinger wakes it:

Handles transactions, if present.
Latches new graphic buffers, if present.
Performs a new composition, if step 1 or 2 resulted in a change to the display contents.
To perform a new composition, SurfaceFlinger creates and destroys layers or modifies layer states, as applicable. It also updates layers with their current contents, using calls such as setLayerBuffer or setLayerColor. After all layers are updated, SurfaceFlinger calls validateDisplay, which tells the HWC to examine the state of the layers and determine how composition will proceed. By default, SurfaceFlinger attempts to configure every layer such that the layer is composited by the HWC; though in some circumstances, SurfaceFlinger composites layers through the GPU fallback.

After the call to validateDisplay, SurfaceFlinger calls getChangedCompositionTypes to see if the HWC wants any of the layer composition types changed before performing the composition. To accept the changes, SurfaceFlinger calls acceptDisplayChanges.

If any layers are marked for SurfaceFlinger composition, SurfaceFlinger composites them into the target buffer. SurfaceFlinger then calls setClientTarget to give the buffer to the display so that the buffer can be displayed on the screen or further composited with layers that haven't been marked for SurfaceFlinger composition. If no layers are marked for SurfaceFlinger composition, SurfaceFlinger bypasses the composition step.

Finally, SurfaceFlinger calls presentDisplay to tell the HWC to complete the composition process and display the final result.

Multiple displays
Android 10 supports multiple physical displays. When designing an HWC implementation intended for use on Android 7.0 and higher, there are some restrictions not present in the HWC definition:

It's assumed that there's exactly one internal display. The internal display is the display that the initial hotplug reports during boot. After the internal display is hotplugged, it can't be disconnected.
In addition to the internal display, any number of external displays may be hotplugged during normal operation of the device. The framework assumes that all hotplugs after the first internal display are external displays, so if any more internal displays are added, they're categorized incorrectly as Display.TYPE_HDMI instead of Display.TYPE_BUILT_IN.
While the SurfaceFlinger operations described above are performed per-display, they're performed sequentially for all active displays, even if the contents of only one display are updated.

For example, if the external display is updated, the sequence is:

```
// In Android 9 and lower:

// Update state for internal display
// Update state for external display
validateDisplay(<internal display>)
validateDisplay(<external display>)
presentDisplay(<internal display>)
presentDisplay(<external display>)

// In Android 10 and higher:

// Update state for internal display
// Update state for external display
validateInternal(<internal display>)
presentInternal(<internal display>)
validateExternal(<external display>)
presentExternal(<external display>)
```

Virtual display composition
Virtual display composition is similar to external display composition. The difference between

virtual display composition and physical display composition is that virtual displays send output to a Gralloc buffer instead of to the screen. Hardware Composer (HWC) writes the output to a buffer, provides the completion fence, and sends the buffer to a consumer (such as the video encoder, GPU, CPU, and so on). Virtual displays can use 2D/blitter or overlays if the display pipeline writes to memory.

Modes
Each frame is in one of three modes after SurfaceFlinger calls the validateDisplay() HWC method:

GLES — The GPU composites all layers, writing directly to the output buffer. The HWC isn't involved in composition.
MIXED — The GPU composites some layers to the framebuffer and HWC composites the framebuffer and the remaining layers, writing directly to the output buffer.
HWC — HWC composites all layers and writes directly to the output buffer.
Output format
Virtual display buffer output formats depend on their mode:

GLES mode — The EGL driver sets the output buffer format in dequeueBuffer(), typically RGBA_8888. The consumer must be able to accept the output format the driver sets or the buffer can't be read.
MIXED and HWC modes — If the consumer needs CPU access, the consumer sets the format. Otherwise, the format is IMPLEMENTATION_DEFINED, and Gralloc sets the best format based on the usage flags. For example, Gralloc sets a YCbCr format if the consumer is video encoder and HWC can write the format efficiently.
Note: Android 10 removes the requirement that eglSwapBuffers() dequeues buffers after rendering begins. Buffers may be dequeued immediately.
Synchronization fences
Synchronization (sync) fences are a crucial aspect of the Android graphics system. Fences let CPU work proceed independently from concurrent GPU work, blocking only when there's a true dependency.

For example, when an app submits a buffer that's being produced on the GPU, it also submits a sync fence object. This fence signals when the GPU has finished writing into the buffer.

The HWC requires that the GPU finish writing buffers before buffers are displayed. Sync fences are passed through the graphics pipeline with buffers and signal when buffers are written. Before a buffer is displayed, the HWC checks if the sync fence has signaled, and if it has, it displays the buffer.

For more information about sync fences see Hardware Composer Integration.

Let’s walk through the **Hardware Composer (HWC) HAL implementation** in a **detailed, but easy-to-understand way**, broken down step-by-step.

# 🔧 What is the Hardware Composer (HWC) HAL?

The **Hardware Composer HAL** is part of Android's graphics pipeline. It helps **offload work from the GPU** by using display hardware to combine layers (e.g., app, wallpaper, status bar) before they're shown on screen.

Instead of the GPU doing all the work of combining (compositing) layers, **HWC can use display-specific hardware (like overlays, blitters)**, which is more efficient.

---

# 🛠️ What Are You Implementing?

As a device or display hardware vendor, when you "implement HWC", you're:

- Writing the code that tells Android how your display hardware can do **layer composition**
- Telling SurfaceFlinger:
  - How many overlays are available
  - Which formats and layer types you support
  - Which layers can be composed by hardware
- Handling layer composition, virtual displays, protected content, etc.

---

# 🧩 Key Building Blocks of HWC

| Term | Meaning |
|---|---|
| **Layer** | One visual component (e.g., app, status bar, nav bar) |
| **Display** | A screen (physical or virtual) |
| **Composer HAL** | The new HIDL interface for HWC (Android 8.0+) |
| **hwcomposer2.h** | Legacy C interface (vendors can still use this with Composer HAL passthrough) |
| **VSync** | A signal that tells when to update the screen (once per frame) |
| **Sync Fence** | A "signal" used to track when a buffer is ready to use |

---

# 🧭 How the Graphics Pipeline Flows (With HWC)

1. **App draws content** to a buffer (via GPU)
2. **SurfaceFlinger** collects layers (app, status bar, etc.)
3. **SurfaceFlinger** calls **`validateDisplay()`** on HWC:
   - "Can you handle these layers?"

4. HWC responds:
    - "I'll take layer A and C" (hardware)
    - "Use GPU for B" (client composition)
5. SurfaceFlinger:
    - Uses GLES (GPU) to blend layers the HWC can't handle
    - Sends final buffer to HWC
6. SurfaceFlinger calls:
    - `setClientTarget()`: tells HWC which GPU-blended buffer to use
    - `presentDisplay()`: HWC shows the frame on screen

---

# 🔨 Steps to Implement HWC (Simplified Roadmap)

## ✅ Step 1: Start with a Dummy (GLES-only)

- Implement all functions as stubs
- Let GPU do all composition
- Always respond to `validateDisplay()` with: "Let GPU do everything"

## ✅ Step 2: Incremental HWC Composition

- Start supporting some layers (e.g., app + nav bar)
- In `validateDisplay()`:
    - Mark the first few layers as `HWC_COMPOSITION_DEVICE`
    - Others as `HWC_COMPOSITION_CLIENT`

## ✅ Step 3: Optimize Composition

- Add logic to:
    - Handle large backgrounds (e.g., scrolling wallpapers)
    - Detect when screen content isn't changing (reuse previous frames)
    - Offload protected video
- Consider use cases like:
    - Home screen
    - Full-screen video playback
    - Split screen

---

# 🧱 Core Concepts You Must Support

| Feature | Description |
|---|---|
| **Overlays** | At least 4 for app, wallpaper, nav bar, status bar |
| **Alpha blending** | Mix transparency (e.g., status bar over app) |
| **Protected content** | Handle DRM-protected video securely |
| **YUV formats & tiling** | Handle video-specific formats |
| **Virtual Displays** | Compose output to a buffer (e.g., for screen recording or casting) |

---

# 🧩 Composer HAL: HIDL vs Legacy

## 1. New (Preferred): Composer HAL (HIDL-based)

- Implements functions like `createLayer()`, `presentDisplay()` as HIDL methods
- SurfaceFlinger uses Binder IPC to call them

## 2. Legacy (Still supported): hwcomposer2.h

- You implement C functions like `HWC2_PFN_CREATE_LAYER`
- Composer HAL uses function pointers to call into your implementation

---

# 📜 How a Frame is Drawn (Every VSync)

1. **App submits buffer** (with sync fence)
2. **SurfaceFlinger wakes up** on VSync
3. SurfaceFlinger:
    - Applies transactions (e.g., resizing, adding/removing layers)
    - Latches new buffers
    - Calls `validateDisplay()` on HWC
    - Receives list of layers to offload to HWC vs GPU
    - Calls `acceptDisplayChanges()`
4. If GPU is involved:
    - SurfaceFlinger renders GPU-composed layers into a buffer
    - Calls `setClientTarget()` with this buffer
5. Finally, SurfaceFlinger calls `presentDisplay()`:

- Tells HWC to show the final image
- HWC waits on sync fences before showing buffers

## 📺 Support for Multiple Displays

- Android assumes one **internal** display (main screen)
- External displays (e.g., HDMI) can be hotplugged
- Virtual displays go to memory (not real screens), often for:
  - Screen recording
  - Wireless display
  - Video encoder

## 🧪 Composition Modes

| Mode | Who Composes What |
|---|---|
| **GLES** | GPU does everything (slowest, fallback mode) |
| **HWC** | HWC does everything (best performance & battery) |
| **MIXED** | GPU does some layers; HWC finishes composition |

## 🧷 Sync Fences (Very Important)

- A **fence** is a signal: “Is the buffer ready?”
- When GPU draws a buffer, it adds a **fence**
- HWC **waits for that fence** before using the buffer
- Avoids tearing, glitches, and memory corruption

## ✅ Summary Checklist for Your HWC Implementation

| Requirement | Mandatory for Modern HWC |
|---|---|
| Basic layer support | ✅ |
| Composition via overlays/blitters | ✅ |
| Fallback to GLES | ✅ |
| Virtual display support | ✅ |
| Protected content | ✅ |

| Requirement | Mandatory for Modern HWC |
|---|---|
| Sync fence handling | ✅ |
| Multiple display support | ✅ |
| HIDL-based Composer HAL interface | ✅ (Android 8+) |

---

Would you like a **code-level walkthrough** (e.g., how to implement `validateDisplay` or `createLayer`) or a **diagram** showing the full graphics pipeline with HWC?

Hotplug handling

bookmark_border
Display capabilities (such as display modes and supported HDR types) can change dynamically on devices that have externally connected displays (with HDMI or DisplayPort), such as Android TV set-top-boxes (STBs) and over-the-top (OTT) devices. This change can happen as a result of an HDMI hotplug signal, such as when the user switches from one display to another or boots the device without a connected display. Android 12 and higher includes changes in the framework to handle hotplugging and dynamic display capabilities.

This page describes the handling of display hotplugs and changes in display capabilities in the Composer HAL implementation. Additionally it discusses how to manage the associated framebuffer and prevent race conditions in these situations.

Note: The term display mode used in the Android API and framework, is equivalent to the term display config used in the Composer HAL.
Update display capabilities
This section describes how the Android framework handles changes in display capabilities initiated by Composer HAL.

Before Android can handle changes in display capabilities properly, the OEM must implement Composer HAL such that it uses onHotplug(display, connection=CONNECTED) to notify the framework of any changes to display capabilities. After that's implemented, Android handles changes to display capabilities as follows:

On detecting a change in display capabilities, the framework receives an onHotplug(display, connection=CONNECTED) notification.
On receiving the notification, the framework drops its display state and recreates it with the new capabilities from the HAL by using the getActiveConfig, getDisplayConfigs, getDisplayAttribute, getColorModes, getHdrCapabilities, and getDisplayCapabilities methods.
After the framework recreates a new display state, it sends the onDisplayChanged callback to the apps that are listening for such events.
Note: Ensure that getDisplayConfigs accurately reports all supported display refresh rates and resolutions.

The framework reallocates the framebuffers on subsequent onHotplug(display, connection=CONNECTED) events. See Client framebuffer management for more information on how to properly manage framebuffer memory to avoid failures during allocation of new framebuffers.

Handle common connection scenarios
This section covers how to properly handle various connection scenarios in your implementations when the primary display is connected and disconnected.

Having been built for mobile devices, the Android framework doesn't have built-in support for a disconnected primary display. Instead the HAL must replace the primary display with a placeholder display in its interactions with the framework in the case when a primary display is physically disconnected.

Warning: Don’t send an onHotplug(display, connection=DISCONNECTED) notification for the primary display, as this causes the framework to crash.
The following scenarios can occur in STBs and TV dongles that have externally connected displays that can be disconnected. To implement support for these scenarios, use the information in the table below:

Scenario Handling
No connected display at boot time
Send an onHotplug(display, connection=CONNECTED) signal from the Composer HAL to the framework.
Replace the physical display state inside the Composer HAL with a placeholder display state.
Note: We recommend the placeholder display to have a single supported mode with resolution of 1080x1920 and refresh rate of 60 Hz, as this display mode is supported by most apps.
Primary display is physically connected
Send another onHotplug(display, connection=CONNECTED) event from the Composer HAL to the framework.
This causes the framework to reload all display capabilities.

Primary display is physically disconnected
Send another onHotplug(display, connection=CONNECTED) event from the Composer HAL to the framework.
Replace the physical display state inside the Composer HAL with a placeholder display state.
The placeholder display must have a single display mode, so that the framework sends the onDisplayChanged callback to apps (because the set of supported modes have changed). This single display mode must match the last active mode of the physical display before disconnection, so that apps don’t receive configuration change events.
Non-HDMI connection considerations
Android TV only supports the following resolutions:

720x1280
1080x1920
2160x3840

4320x7680
When an STB or TV dongle attempts to display an unsupported resolution, such as 480i over a CVBS connection, an error message is presented to the user.

If the STB or TV dongle has both HDMI and non-HDMI connections, the HDMI connection is the primary display and the non-HDMI connection is inactive. As a result, if the HDMI connection is disconnected while the non-HDMI connection is still connected, an event is sent to SurfaceFlinger and the capabilities of the non-HDMI display must be reflected through getDisplayAttribute and other iComposerClient APIs (such as getHdrCapabilities).

Use sequential config IDs to prevent race conditions
Race conditions can arise if the Composer HAL updates the supported display configs concurrently with the framework calling setActiveConfig or setActiveConfigWithConstraints. The solution is to implement Composer HAL to use sequential IDs and prevent this problem.

This section describes how the race conditions might occur, followed by details on how to implement Composer HAL so that it uses sequential IDs to prevent such conditions.

Consider the following sequence of events, when new, sequential IDs are NOT assigned to the new display configs, causing a race condition:

The supported display config IDs are:

id=1, 1080x1920 60 Hz
id=2, 1080x1920 50 Hz
The framework calls setActiveConfig(display, config=1).

Concurrently, the Composer HAL processes a change of display configs and updates its internal state to a new set of display configs, shown as follows:

id=1, 2160x3840 60 Hz
id=2, 2160x3840 50 Hz
id=3, 1080x1920 60 Hz
id=4, 1080x1920 50 Hz
Composer HAL sends an onHotplug event to the framework, to notify that the set of supported modes has changed.

The Composer HAL receives setActiveConfig(display, config=1) (from step 2).

The HAL interprets that the framework has requested a config change to 2160x3840 60 Hz, although in reality 1080x1920 60 Hz was desired.

The process using nonsequential ID assignments ends here with a misinterpretation of the desired config change.

Configure Composer HAL to use sequential IDs

To avoid such race conditions, the OEM must implement the Composer HAL as follows:

When the Composer HAL updates the supported display configs, it assigns new, sequential IDs to the new display configs.
When the framework calls setActiveConfig or setActiveConfigWithConstraints with an invalid config ID, the Composer HAL ignores the call.
These steps serve to prevent race conditions as shown in the following discussion.

Consider the following sequence of events, when new, sequential IDs are assigned to the new display configs:

The supported display config IDs are:

id=1, 1080x1920 60 Hz
id=2, 1080x1920 50 Hz
The framework calls setActiveConfig(display, config=1).

When a change of display configs is processed, the next set of config IDs are assigned starting from the next unused integer, shown as follows:

id=3, 2160x3840 60 Hz

id=4, 2160x3840 50 Hz

id=5, 1080x1920 60 Hz

id=6, 1080x1920 50 Hz

The Composer HAL sends an onHotplug event to the framework, to notify that the set of supported modes has changed.

The Composer HAL receives setActiveConfig(display, config=1) (from step 2).

The Composer HAL ignores the call as the ID is no longer valid.

The framework receives and processes the onHotplug event from step 4. It calls into the Composer HAL using the functions getDisplayConfigs and getDisplayAttribute. With these functions the framework identifies the new ID (5) for the desired resolution and refresh rate of 1080x1920 and 60 Hz.

Note: The framework also calls getActiveConfig to get the current display mode and if it differs from the desired display mode, it calls setActiveConfig() again.
The framework sends another setActiveConfig event with an updated ID of 5.

The Composer HAL receives setActiveConfig(display, config=5) from step 5.

The HAL correctly interprets that the framework has requested a config change to 1080x1920 60 Hz.

As shown in the example above, the process using sequential ID assignments ensures that the race condition is prevented and the correct display config change is updated.

### 🔌 Hotplug Handling in Composer HAL (Android 12+) — Detailed Explanation

Hotplugging refers to the dynamic connection or disconnection of external displays (e.g., HDMI/DisplayPort). Android 12+ enhances framework support for hotplugging and changing display capabilities **without crashing or misbehaving**. Below is a structured explanation of what needs to happen and why — both **in the HAL** and **in the framework**.

---

## 🔧 1. How Hotplug Events Are Handled

### 🧠 *Basic Idea*:

When a display (e.g., HDMI) is plugged in or unplugged, the **Composer HAL** must notify the **Android framework** using **`onHotplug(display, connection=CONNECTED|DISCONNECTED)`**. This tells SurfaceFlinger and the framework to reconfigure the display stack.

---

## 🔄 2. Updating Display Capabilities

When a hotplug signal is detected (e.g., new HDMI device), the HAL must:

1. Call:

```cpp
onHotplug(display_id, connection=CONNECTED)
```

2. Framework reacts by:
   - Dropping old display state.
   - Re-creating it by calling:
     - **`getActiveConfig`**
     - **`getDisplayConfigs`**
     - **`getDisplayAttribute`**
     - **`getHdrCapabilities`**
     - **`getColorModes`**

- **`getDisplayCapabilities`**

3. After reinitialization, the framework notifies apps by sending:

```java
onDisplayChanged()
```

> 📌 *Important:* Your HAL must ensure these methods return correct and complete information or app rendering and power optimizations may break.

---

## 🧩 3. Handling Special Scenarios

### ⚠️ *Rule of Thumb: Never disconnect the primary display.*

| Scenario | What to Do in HAL |
|---|---|
| **No HDMI display at boot** | Replace with **placeholder display** (e.g., 1080x1920 @ 60 Hz). Send **`onHotplug(..., CONNECTED)`**. |
| **HDMI connected after boot** | Send another **`onHotplug(..., CONNECTED)`** with real capabilities. |
| **HDMI disconnected** | Do **not** send **`DISCONNECTED`**. Instead:<br>- Replace with placeholder state.<br>- Send **`onHotplug(..., CONNECTED)`** again to reset capabilities.<br>- Ensure placeholder matches **last active config** to avoid app-side configuration changes. |

---

## 📺 4. Managing Virtual and Physical Displays

- **Virtual displays**: SurfaceFlinger calls **`createVirtualDisplay()`**; output goes to a **`Gralloc`** buffer (not screen).
- **Physical displays**: Handle via **`onHotplug()`**, including HDMI, DisplayPort.

> 📌 For Android TV STBs or OTT devices: HDMI is considered the **primary** display.
> Disconnecting HDMI must still leave a valid, minimal display state in place (e.g., fallback to 1080p placeholder).

---

## 🧱 5. Avoiding Race Conditions with Display Config IDs

When display capabilities change (like resolution or refresh rate), the HAL must assign **new, sequential config IDs**. This avoids bugs where:

- The framework sends **`setActiveConfig(config_id=1)`** based on old configs.

- But the HAL has reused ID `1` for a *different* resolution in the new list.
- Result: **Wrong resolution is applied.**

### ✅ *Correct Way Using Sequential IDs*:

**Before:**

```text
id=1 → 1080p60
id=2 → 1080p50
```

**After:**

```text
id=3 → 2160p60
id=4 → 2160p50
id=5 → 1080p60
id=6 → 1080p50
```

> 🔐 The old IDs (1,2) are never reused. New configs start from the next unused integer.

### 🔄 If framework sends stale ID:

- HAL **ignores** it.
- Framework re-queries with:
  - `getDisplayConfigs`
  - `getDisplayAttribute`
- Framework re-sends updated `setActiveConfig(config_id=5)` (for example).
- HAL applies correct config.

---

## 💡 6. Framebuffer Reallocation

After hotplug or config change, the framework reallocates framebuffers. Your HAL must:

- Properly manage memory.
- Avoid stale framebuffer handles.
- Prevent allocation failures (e.g., by releasing old buffers).

---

## 📦 7. Best Practices Summary

| Area | Recommendation |
|---|---|
| **Primary Display** | Never send **`DISCONNECTED`**. Always replace with placeholder if removed. |
| **Display Configs** | Always assign new, sequential config IDs. Never reuse. |
| **Display Info** | Implement **`get*()`** methods to accurately reflect supported capabilities. |
| **HDMI + non-HDMI** | Always treat HDMI as primary; fallback to others only if HDMI is disconnected. |
| **Race Conditions** | Prevent with config ID strategy; ignore unknown/stale config IDs. |

---

# ✅ Example Placeholder Config

```cpp
// Placeholder display returned when HDMI is disconnected
{
  resolution = 1080x1920,
  refresh_rate = 60Hz,
  config_id = N+1
}
```

---

Let me know if you'd like:

- A **sample implementation sketch** of these hotplug handlers in HIDL/Composer HAL
- Details on **VSync callback**, **`presentDisplay()`**, or **`validateDisplay()`**
- Diagrams to visualize this architecture

yes

Great! Let's now **break down the communication and roles of the Android graphics pipeline** in **simple and clear** steps — focusing on how the app, **`Surface`**, **`SurfaceView`**, **`SurfaceFlinger`**, **`BufferQueue`**, **`HWC`**, and others work together, including the role of **Hardware Composer HAL (HWC)**.

---

# 🔄 HIGH-LEVEL FLOW: Who Talks to Whom?

## 1. App

- The app (e.g., your camera app or game) draws things using OpenGL ES or the Android UI toolkit.
- It needs a surface to draw on.

## 2. SurfaceView / SurfaceTexture / Surface

- These are frontend objects that **provide a Surface** (which is backed by a **BufferQueue**).
- A **`Surface`** is the **producer** side of a **`BufferQueue`** — your app writes here.

### 3. BufferQueue

- Acts like a pipe:
  - **Producer**: App writes graphic buffers.
  - **Consumer**: System reads from them (e.g., SurfaceFlinger).
- It supports multiple buffers to allow smooth rendering without stalling.

### 4. SurfaceFlinger

- Android's **display composer**. It reads buffers from all visible apps.
- It is the **consumer** of the app's **`BufferQueue`**.
- Combines (composites) all layers (status bar, nav bar, app UI) into a final screen image.
- Decides whether to:
  - Use GPU (OpenGL/GLES composition)
  - Use hardware overlays (via HWC HAL)

### 5. Hardware Composer HAL (HWC)

- A vendor-provided module that tells SurfaceFlinger:
  - "This layer can be composed directly using hardware"
  - Or "Please use GLES to compose this"
- It talks directly to **display hardware**.

---

## 🧠 DETAILED EXAMPLE: Drawing a Button on Screen

Let's say your app has a button in a SurfaceView:

1. **App**:
   - Creates a **`SurfaceView`**, which internally creates a **`Surface`** with a **`BufferQueue`**.
   - Starts drawing a button via Canvas or OpenGL on that **`Surface`** (i.e., producing to BufferQueue).
2. **BufferQueue**:
   - Holds the rendered frame from the app.
   - Notifies **`SurfaceFlinger`** that a new frame is ready.
3. **SurfaceFlinger**:

- Reads that buffer (consumes it).
- Combines it with other system UI layers (e.g., status bar).
- Asks **`HWC`**: "Can you render this directly with hardware?"

4. **HWC HAL**:
    - Replies: "This layer can go directly to an overlay," or "Use GLES to blend these and give me the result."
5. **Display**:
    - Gets the final output — either directly from hardware overlays or a GPU-composed buffer.

---

## 🧰 Bonus: Role of DMA-BUF, ION, DMA-HEAP

- **GraphicBuffer** objects behind a Surface use memory from the system:
    - Older Androids used **ION** to allocate memory for these buffers.
    - Newer ones use **DMA-HEAP** (a modern replacement).
- These buffers are shared across processes (e.g., app ↔ SurfaceFlinger ↔ HWC) using **DMA-BUF file descriptors**.
- These allow **zero-copy**, efficient sharing of large image frames between components.

---

## 🎥 Surface vs SurfaceView vs SurfaceTexture

| Item | Role |
|---|---|
| **`Surface`** | The low-level drawing target (used by both SurfaceView/SurfaceTexture) |
| **`SurfaceView`** | A View that hosts a Surface — used for regular UI drawing |
| **`SurfaceTexture`** | Used for textures, camera preview, or OpenGL — backed by Surface too |

---

## 🧩 Summary Chart

```text
[App]
   |
   | writes to
   v
[Surface] —> [BufferQueue] —> [SurfaceFlinger] —> [HWC HAL] —> [Display]
                        ^                   |
                        | reads buffer      | asks about composition
                        |                   v
                  [GraphicBuffer]    [GLES or Overlays]
```

---

Would you like to also see **step-by-step logs**, **code snippets**, or how it looks in **Winscope trace**?

AIDL for Hardware Composer HAL

bookmark_border
Starting in Android 13, the Hardware Composer (HWC) HAL is defined in AIDL and the HIDL versions ranging from android.hardware.graphics.composer@2.1 to android.hardware.graphics.composer@2.4 are deprecated.

This page describes the differences between the AIDL and the HIDL HAL for the HWC and the implementation and testing of the AIDL HAL.

Because of the advantages offered by AIDL, vendors are encouraged to implement the AIDL composer HAL starting Android 13 instead of the HIDL version. See the Implementation section for more information.

Differences between AIDL and HIDL HALs
The new AIDL composer HAL, named android.hardware.graphics.composer3, is defined in IComposer.aidl. It exposes an API similar to the HIDL HAL android.hardware.graphics.composer@2.4 with the following changes:

Removal of the Fast Message Queue (FMQ) in favor of parcelable commands.

The AIDL HAL defines the command interface based on strongly typed parcelable types as opposed to the serialized commands over FMQ in HIDL. This provides a stable interface for commands and a more readable definition of how the command payload is interpreted.

The executeCommands method is defined in IComposerClient.aidl as

CommandResultPayload[] executeCommands(in DisplayCommand[] commands);
where each command is a strongly typed parcelable type defined in DisplayCommand.aidl. Command responses are strongly typed parcelables defined in CommandResultPayload.aidl.

Removal of IComposerClient.getClientTargetSupport as there are no active clients for this method.

Representation of colors as floats instead of bytes to better align with the upper graphics stack in Android as defined in ASurfaceTransaction_setColor.

Addition of new fields for controlling HDR content.

In the AIDL HAL, mixed SDR/HDR layer stacks support the seamless dimming of SDR layers when an HDR layer is simultaneously on screen.

The brightness field in LayerCommand lets SurfaceFlinger specify a per-layer brightness, so that the HWC dims the layer's content in linear light space, as opposed to gamma space.

The brightness field in ClientTargetPropertyWithBrightness lets the HWC specify the brightness space for client composition and to instruct RenderEngine whether to dim SDR layers in client composition.

The dimmingStage field lets the HWC configure when RenderEngine should dim content. This accommodates vendor-defined ColorModes, which might prefer to dim in gamma space, to allow vendor-defined contrast enhancements in their color pipelines.

Addition of a new composition type DISPLAY_DECORATION in Composition.aidl for screen decorations.

Some devices have dedicated hardware to optimize drawing the alpha mask that smooths rounded corners and cutouts on displays. Devices with such hardware must implement IComposerClient.getDisplayDecorationSupport to return a DisplayDecorationSupport structure as defined in the new DisplayDecorationSupport.aidl. This structure describes the PixelFormat and AlphaInterpretation enums required by the device. Upon this implementation, System UI marks the alpha mask layer as DISPLAY_DECORATION, a new composition type that takes advantage of the dedicated hardware.

Addition of a new expectedPresentTime field to DisplayCommand.aidl.

The expectedPresentTime field lets SurfaceFlinger set the expected present time to when the current content must be displayed on screen. With this feature, SurfaceFlinger sends a present command to the implementation ahead of time, allowing it to pipeline more of the composition work.

Addition of new APIs to control boot display configuration.

Using BOOT_DISPLAY_CONFIG, vendors can specify that the boot display configuration is supported. The setBootDisplayConfig, clearBootDisplayConfig, and getPreferredBootDisplayConfig methods use BOOT_DISPLAY_CONFIG as follows:

Using setBootDisplayConfig, the framework informs vendors of the boot time display configuration. Vendors must cache in the boot display configuration, and boot in this config on next reboot. If the device is unable to boot in this config, the vendor must find a config that matches the resolution and refresh rate of this config. If no such config exists, the vendor should use their preferred display config.

Using clearBootDisplayConfig, the framework informs the vendors to clear the boot display configuration, and boot in their preferred display config during the next reboot.

Using getPreferredBootDisplayConfig, the framework queries the vendor's preferred boot mode.

When the boot display configuration isn't supported, these methods return a value of UNSUPPORTED.

Addition of new APIs to control the display idle timer.

Using DISPLAY_IDLE_TIMER, vendors can specify that an inactivity timer is implemented by the vendor for this display. When idle, this capability changes the refresh rate to a lower setting to preserve power. The platform uses setIdleTimerEnabled to control the timeout of the timer, and in some cases, to disable it in order to prevent undesired refresh rate switches when idle.

Using the IComposerCallback.onVsyncIdle callback indicates to the platform that the display is idle and the vsync cadence has changed. The platform responds to this callback by resetting its vsync model. It forces a vsync resync on the next frame, and learns the new vsync cadence.

Implementation
Vendors aren't required to implement the AIDL HAL for Android 13. However, they're encouraged to implement the AIDL composer HAL instead of the HIDL version to use the new functionality and APIs.

A reference implementation for the AIDL HWC HAL is implemented in Android emulators.

Testing
To test your implementation, run VtsHalGraphicsComposer3_TargetTest.

Great — let's walk through what this **AIDL-based Hardware Composer HAL (HWC HAL)** means and how it differs from the older HIDL version, explained simply and clearly.

---

# 🧠 What is the AIDL HWC HAL?

The **Hardware Composer HAL** is a module that helps Android decide **how to show frames on screen** — whether using GPU or directly using hardware overlays (for better performance and lower power).

### Previously:

It used **HIDL (HAL Interface Definition Language)**.

### Now (Android 13+):

It uses **AIDL (Android Interface Definition Language)** — a newer, cleaner, and easier way to define HALs.

---

# 🎯 Why Switch from HIDL to AIDL?

| Feature | HIDL HWC HAL | AIDL HWC HAL |
| --- | --- | --- |
| Data communication | Uses FMQ (Fast Message Queue) | Uses typed **parcelables** (simpler) |
| Interface stability | Binary safe but complex | **Readable, stable**, and safer |
| Color handling | Uses bytes (less accurate) | Uses **floats** for precision |
| HDR support | Basic | Enhanced: SDR/HDR blending supported |
| Display decorations | Not supported | Supports special decoration layers |
| Boot display config APIs | Not available | **New** APIs to control boot display |
| Display idle timer control | Not available | **New** APIs to save power |

---

# 🧩 Key Concepts Explained Simply

## 1. No More FMQ — Use Parcelable Commands

Before, commands were serialized into a shared queue (FMQ). Now:

- Each command is a **clearly typed object**.
- This makes the API **easier to read, safer, and less error-prone**.

```aidl
// One command, one object
DisplayCommand {
   SetLayerBrightness brightness;
   SetClientTarget target;
   ...
}
```

## 2. Per-Layer Brightness and HDR Blending

- Each layer can have its **own brightness level**.
- SDR layers (e.g. app UI) can **automatically dim** when HDR content is shown — for better color fidelity.

  Dimming is now done in **linear light space**, not gamma space — for more accurate blending.

---

## 3. New Composition Type: DISPLAY_DECORATION

- This is for **rounded corners**, **notches**, or **cutout effects**.
- Some devices have **special hardware** to draw those decorations efficiently.

- This allows SystemUI to mark such layers for **optimized rendering**.

---

### 4. expectedPresentTime Field

- Allows SurfaceFlinger to **schedule composition earlier**.
- Helps **pipeline** the rendering steps better — which improves **performance and reduces latency**.

---

### 5. Boot Display Config APIs

- New APIs allow Android to:
  - **Set** a specific boot resolution (e.g., for TVs or fixed-size displays).
  - **Clear** it or **query** the preferred one.
- Very useful for **TVs, STBs**, or **multi-display devices**.

---

### 6. Display Idle Timer

- Some displays can **reduce refresh rate** when idle (e.g., 60Hz → 30Hz) to save battery.
- AIDL HWC lets the system:
  - Enable/disable the idle timer.
  - Respond to `onVsyncIdle()` to adjust the vsync timing.

---

## ✅ Testing Your AIDL HWC Implementation

Google provides a VTS test:

```bash
VtsHalGraphicsComposer3_TargetTest
```

This checks whether your HWC AIDL implementation:

- Properly responds to display commands
- Handles composition types
- Respects expectedPresentTime
- Supports boot display config
- Handles idle timer and vsync signals correctly

---

## 🔚 Summary

| Feature | Benefit |
|---|---|
| **AIDL over HIDL** | Easier to understand, safer |
| **Typed commands** | No parsing bugs or ambiguity |
| **HDR/SDR blending** | Higher visual quality |
| **Idle timer & boot config APIs** | Power efficiency + boot flexibility |
| **Display decorations** | Optimized rendering of corners/cutouts |

---

Would you like to see how a **vendor might implement this AIDL HWC HAL**, or how SurfaceFlinger uses it step-by-step?

Client framebuffer management

bookmark_border
Starting with Android 13, new framebuffers, used during client composition, are allocated whenever the display resolution changes. This allocation is performed by SurfaceFlinger on the next invalidate cycle after a resolution change.

Framebuffer management during resolution switches
Resolution changes occur due to one of the following two scenarios:

A hotplug event, initiated by Hardware Composer (HWC), which occurs when swapping from one external display to a different external display that has a different default resolution.

During a hotplug event, the handles to the old framebuffers are released when the old display data is deallocated.

A display mode switch initiated by SurfaceFlinger, which occurs when the user changes the resolution with user settings, or an app changes the resolution with preferredDisplayModeId.

During a display mode switch, the handles to existing client framebuffers are released by SurfaceFlinger before calling setActiveConfig or setActiveConfigWithConstraints.

To avoid catastrophic problems, such as memory fragmentation, on devices that don't reserve enough memory for the old and new framebuffers, it's critical that HWC ceases to use the old framebuffers and releases any handles to these framebuffers as shown in the following cases:

For hotplug events, immediately before calling onHotplug.

For mode switches, immediately after the call to setActiveConfig or setActiveConfigWithConstraints.

Releasing the handles allows the framebuffer memory to be fully deallocated prior to the allocation of new framebuffers that SurfaceFlinger performs during the next invalidate cycle.

Recommendations for framebuffer management
If HWC doesn't release handles to old framebuffers in time, the new framebuffer allocation takes place before the old framebuffer deallocation. This can cause catastrophic problems when the new allocation fails due to fragmentation or other issues. Even worse, if HWC doesn't release these handles at all, a memory leak can occur.

To avoid catastrophic allocation failures, follow these recommendations:

If HWC needs to continue using the old client framebuffers until the new client framebuffers are provided, then it's critical to reserve enough memory for both the old and new framebuffers, and possibly run defragmentation algorithms on the framebuffer memory space.

Allocate a dedicated memory pool for the framebuffers that's separate from the rest of the graphic buffer memory. This is important because between deallocation and reallocation of the framebuffers, a third-party process can attempt to allocate graphics memory. If the same graphics memory pool is used by the framebuffer and if the graphics memory is full, the third-party process can occupy the graphics memory previously allocated by a framebuffer, thus leaving insufficient memory for the framebuffer reallocation or, possibly fragmenting the memory space.

Test framebuffer management
OEMs are advised to test for proper client framebuffer memory management across resolution switches for their device, described as follows:

For hotplug events, simply unplug and reconnect two different displays with different resolutions.

For mode switches, use the ModeSwitchingTestActivity CTS Verifier test to initiate a mode switch for testing framebuffer memory behavior. This test can visually identify problems that are hard to detect programmatically.

This Android 13+ feature explanation describes **client framebuffer management** and how **`SurfaceFlinger`** and **`Hardware Composer (HWC)`** coordinate to manage framebuffers safely when **display resolution changes**. Let's break it down **step by step in simple words**.

---

## 🧠 What is a Framebuffer?

A **framebuffer** is memory that holds the **final image** to be shown on the screen. When an app draws something, it eventually goes into the framebuffer, and then the display reads from it.

# 🧩 What's Changing in Android 13?

When the **display resolution changes** (for example, switching from Full HD to 4K):

- New framebuffers are needed because their **size depends on resolution**.
- These are created by `SurfaceFlinger`.
- But memory is limited. If we don't clean up old ones first, it can lead to **out-of-memory errors** or **memory fragmentation** (when memory is in chunks and can't be reused properly).

---

# 🎬 When do resolution changes happen?

There are two main ways:

### 1. 🔌 Hotplug Event (from HWC)

- Example: You unplug a monitor and plug in a different one with a different resolution.
- HWC (Hardware Composer) detects it.
- It must **release old framebuffer memory** before calling `onHotplug`.

### 2. 🧩 Display Mode Switch (from SurfaceFlinger)

- Example: You or an app manually change screen resolution (e.g., using `preferredDisplayModeId`).
- SurfaceFlinger releases old framebuffer handles **before** calling:
  - `setActiveConfig()` or
  - `setActiveConfigWithConstraints()`

---

# ⚠️ What can go wrong?

### ❌ 1. HWC doesn't release old framebuffer handles in time

- SurfaceFlinger tries to allocate new framebuffer.
- Old one is still held in memory.
- 🔥 Results in **allocation failure or memory leak**.

### ❌ 2. Same memory pool is shared

- Between old framebuffer and general graphics usage.

- While old framebuffer is deallocated, a 3rd-party process (e.g., game or video app) may quickly take that memory.
- When SurfaceFlinger tries to allocate the new framebuffer, there is **no room left** or it's fragmented.

---

# ✅ Recommendations (For OEMs and HWC developers)

### ✔️ Always release framebuffer handles immediately:

- For hotplug: before calling `onHotplug`.
- For mode switch: after calling `setActiveConfig`.

### ✔️ Reserve memory for both old + new framebuffers if needed:

- Useful if you need overlap while switching.

### ✔️ Use a dedicated memory pool for framebuffers:

- So that 3rd party apps don’t steal framebuffer memory during switch.

---

# 🧪 How to test framebuffer memory management

OEMs should test:

### 🔌 For hotplug:

- Connect 2 displays (e.g., HDMI monitors) with different resolutions.
- Plug/unplug and watch behavior.

### 🔄 For mode switch:

- Use **CTS Verifier's ModeSwitchingTestActivity**.
- It will help detect:
  - Allocation failures.
  - Visual glitches (like screen tearing or freeze).

---

# 📌 Summary in Simple Diagram:

```vbnet
App → Surface → SurfaceFlinger → Framebuffer → Display

When resolution changes:
1. SurfaceFlinger releases old framebuffer.
2. HWC must stop using it.
3. New framebuffer is allocated.
4. Display shows content from new framebuffer.
```

Let me know if you want a **visual diagram** or want to understand how **SurfaceFlinger and HWC communicate** using HAL in this flow.

is framebuffer output of surfaceflinger

Yes — **SurfaceFlinger writes to the framebuffer**, but here's the clear explanation:

---

## ✅ What is a framebuffer?

- A **framebuffer** is a **chunk of memory** that holds the final pixels to be shown on the screen.
- Think of it as a "canvas" that the **display hardware** scans and sends to your screen.

---

## ✅ Who writes to the framebuffer?

- The **SurfaceFlinger** is Android's **system compositor**. It collects graphical layers (from apps, system UI, etc.) and **composites** them into a single image.
- After compositing:
  - SurfaceFlinger **writes the final output** to the **framebuffer**.
  - The **display engine (Display Controller / HWC)** reads this and sends it to the screen.

---

## ✅ When is SurfaceFlinger NOT writing to framebuffer?

Sometimes, SurfaceFlinger lets the **Hardware Composer (HWC)** handle everything:

- If HWC can **directly compose layers** (hardware overlays), it **bypasses SurfaceFlinger's GPU composition**.
- In that case, HWC writes directly to **framebuffer** or display interface.

But when:

- Multiple layers need blending.

- Secure content is involved.
- HWC can’t handle a layer.

→ Then **SurfaceFlinger uses GPU (client composition)** and writes to framebuffer itself.

---

## ✅ Final Summary:

| Component | Writes to Framebuffer? | When? |
|---|---|---|
| SurfaceFlinger | ✅ Yes | During GPU (client) composition |
| HWC (via overlays) | ✅ Yes | During direct (hardware) composition |
| Apps / Surface | ❌ No | They write to **GraphicBuffer**, not framebuffer |

So yes, **framebuffer is the output of SurfaceFlinger**, when it does client composition.

Let me know if you want a diagram or flow for this.

Reduce graphics memory consumption

bookmark_border
In the graphics stack, a per-layer buffer cache sits between Composer HAL and SurfaceFlinger to reduce the overhead associated with sending file descriptors over IPC. Prior to Android 14, this buffer cache wasn't purged when a GraphicBufferProducer disconnects from a SurfaceFlinger GraphicBufferConsumer, such as when a MediaCodec is disconnected from a SurfaceView. Starting with Android 14, you can forcefully purge this buffer cache to reduce graphics memory consumption.

Note: The benefits of this feature are most apparent when a user is starting and stopping videos within an app on a device with a high-resolution display and limited memory, such as a television.
Choose from one of the two following options:

For devices launching with Android 14 and higher, you must implement the new Composer HAL API version 3.2. This option is activated by default and saves the most memory. Devices upgrading to 14 and later can also use this option to achieve full memory benefits.
For devices upgrading to Android 14 for which you don't want to implement Composer HAL 3.2 API, you can enable the backward-compatible option. This option saves almost as much memory as the previous option.
The following two sections explain how to implement each option.

Implement the Composer HAL 3.2 API
To achieve full graphics buffer memory benefits, you must:

Update your Composer HAL implementation to version 3.2.

Process LayerCommand::bufferSlotsToClear by purging buffer cache entries indicated by the slot numbers found in the list.
The Composer HAL 3.2 APIs related to graphic buffer memory, including LayerCommand:bufferSlotsToClear, are in LayerCommand.aidl-.

Enable the backward-compatible option
The backward-compatible memory reduction option replaces a real buffer in the cache slot with a 1x1 placeholder buffer, resulting in memory savings for all purged slots, except for the current active buffer slot. To achieve partial memory saving benefits, enable the backward-compatible option by setting the surface_flinger.clear_slots_with_set_layer_buffer sysprop to true. This sysprop is found in the property_contexts file.

Setting this sysprop requires your Composer HAL implementation to correctly handle multiple setLayerBuffer commands for the same layer in a single present cycle.

Enabling the backward-compatible option has the following affects:

For AIDL HALs: SurfaceFlinger sends multiple LayerCommand instances for a single layer, each with a single BufferCommand. Each BufferCommand contains a 1x1 placeholder buffer handle and a slot number for the cache buffer slot that needs to be purged.

For HIDL HALs: SurfaceFlinger sends multiple SELECT_DISPLAY, SELECT_LAYER, SET_BUFFER commands. These commands contain a 1x1 placeholder buffer handle and a slot number for the cache buffer slot that needs to be purged.

The backward-compatible option might cause the Composer HAL to crash on some devices. You might be able to modify your Composer HAL to solve this issue. The code controlling this behavior is found here:

SurfaceFlinger code for AIDL HALs

SurfaceFlinger code for HIDL HALs

Test graphics buffer cache memory consumption
Tests can't verify whether the cache slots are purged by HAL implementations. However, you can use your debugging tools to monitor graphic buffer usage. As you monitor, you should notice that there are fewer out-of-memory errors in scenarios where multiple different videos are stopped and started in quick succession on YouTube.

VTS tests are available that verify that the HAL implementation is functionally capable of receiving the new API calls (HAL version 3.2+) or multiple setLayerBuffer commands for the backward-compatible implementation. However, this shouldn't be considered sufficient testing for proper functionality, as some devices pass these VTS tests, but fail during real-world use cases.

For new VTS tests, navigate to the following links:

HIDL compatible: GraphicsComposerHidlCommandTest::SET_LAYER_BUFFER_multipleTimes

AIDL 3.1 compatible: GraphicsComposerAidlCommandTest::SetLayerBufferMultipleTimes

AIDL 3.2: GraphicsComposerAidlCommandV2Test::SetLayerBufferSlotsToClear

Let's break this down in **simple terms** and explain **how graphics memory is reduced in Android 14**, especially focusing on **SurfaceFlinger**, **Composer HAL**, and the **buffer cache**.

---

### 🧠 What is the problem?

When playing a video or showing a surface (like a `SurfaceView`) in Android, the graphics system (SurfaceFlinger and HAL) uses **graphics buffers** to render frames.

Each **layer** (like a video, UI, or image) has its own **set of buffers** stored in a cache. These buffers take up **GPU memory (RAM)**.

🖍️ **But the issue is:**

> When that layer (like a video or surface) **goes away** (e.g., user stops video), the old buffers **are not always cleaned up**, especially **before Android 14**.

This leads to **wasted memory**, especially on devices like **TVs** or phones with **high-resolution displays** and **low RAM**.

---

## ✅ Android 14 introduces two ways to fix this:

---

### ✅ Option 1: Full fix using Composer HAL 3.2 (Recommended)

This is the **best and default** solution for devices launching with Android 14+.

#### 📦 What's new?

A new AIDL API: `LayerCommand.bufferSlotsToClear`

#### 🛠️ What must be done?

1. **HAL implementers** (OEMs) update Composer HAL to version **3.2**.
2. The HAL must handle this new command:
   When SurfaceFlinger sends a list of **slot numbers**, HAL **clears** those corresponding buffers from its **per-layer cache**.

🎯 **Effect:**
Old unused GPU buffers are **purged** completely → **saves memory**.

---

## ✅ Option 2: Backward-compatible fix (for older HALs)

This is for devices **upgrading** to Android 14 but **not updating** HAL to version 3.2.

### 🛠️ What to do?

Just set the system property:

```bash
surface_flinger.clear_slots_with_set_layer_buffer=true
```

This makes SurfaceFlinger **send 1x1 dummy buffers** to clear old buffer slots.

### ⚠️ Requirements:

HAL must be able to handle **multiple `setLayerBuffer` calls** for the same layer in a single frame cycle.

### 📉 Drawback:

- Doesn't clear the **current active buffer slot**
- Saves **almost** as much memory, but not 100%
- Some HALs may **crash** if they don't expect dummy buffers

---

## 📊 Summary Table

| Feature | Composer HAL 3.2 | Backward-Compatible |
|---|---|---|
| Memory Savings | ✅ Full | ⚠️ Almost full |
| API Used | **`bufferSlotsToClear`** | **`setLayerBuffer`** with dummy |
| Needs HAL Update | ✅ Yes | ❌ No (just sysprop) |
| Risk of Crashes | ❌ Low | ⚠️ Higher |

---

# 👨‍🔬 Testing and Debugging

- **VTS Tests:** These confirm if HAL can accept the new commands, but **don't guarantee** memory is actually freed.

- **Manual Checks:** Use tools like `dumpsys SurfaceFlinger`, `meminfo`, or `systrace` to check GPU memory usage over time, especially:
  - While opening/closing multiple videos
  - Repeatedly attaching/detaching `SurfaceView` or `MediaCodec`

---

## 📦 Example: Real-world Use Case

You're building a **TV app** (like YouTube):

- The user plays a video (buffers allocated)
- Then switches to another video → old buffers remain
- After 10 switches → app crashes due to **GPU memory exhaustion**

**With HAL 3.2 or backward-compatible clearing**, unused buffers get purged → memory stays low → no crash.

---

Would you like to see how `SurfaceFlinger` sends `LayerCommand.bufferSlotsToClear` in code?

confusing

Let's break it down in **simple terms**, step by step. The goal here is to **reduce graphics memory usage** in Android 14 and above by **clearing unused buffer memory** (like video frames) when they are **no longer needed**.

---

## 🤔 Why is there a problem?

When you play and stop videos repeatedly (e.g. in YouTube), the memory used for video frames (**graphic buffers**) isn't always released.
This unused memory builds up in a **buffer cache** between:

- **SurfaceFlinger** (the system compositor), and
- **Composer HAL** (hardware-specific graphics code).

Especially on **low-memory, high-resolution** devices (like TVs), this can lead to **out-of-memory (OOM)** problems.

---

## ✅ What's the solution in Android 14?

Android 14 adds a way to **clear old graphic buffers** from the cache **when they're no longer needed**.

There are **2 ways** to do this depending on your device:

---

# 🟢 Option 1: New Devices (or full upgrade) — Use Composer HAL 3.2

### 📌 What to do:

- Update your **Composer HAL implementation** to **version 3.2** (if you build the OS).
- Handle a new command from SurfaceFlinger:
  ➤ **`LayerCommand::bufferSlotsToClear`**
  This tells you **which buffer slots to clear**.

### ✅ Result:

- **Full memory savings**.
- Buffers are properly freed from memory.
- Best performance.

---

# 🟡 Option 2: Old Devices — Use Backward-Compatible Mode

This is for devices **upgrading to Android 14** but **not updating Composer HAL**.

### 📌 What to do:

- Set this system property:

```ini
surface_flinger.clear_slots_with_set_layer_buffer = true
```

- Your Composer HAL must support receiving **multiple setLayerBuffer() calls** on the same layer.

### 🤔 What actually happens:

- SurfaceFlinger sends **fake 1x1 "dummy" buffers** to **overwrite** the real large buffer.
- This **frees memory**, because the big buffer is no longer referenced.

---

### 🚨 Gotchas

- This **may crash** Composer HAL if it doesn't handle multiple buffers correctly.
- Needs HAL-side testing.
- VTS tests exist to check basic compatibility, but they are **not enough**—real testing on apps like YouTube is needed.

---

## 🧪 How to test it works?

- Use memory debugging tools (**`dumpsys`**, **`meminfo`**, etc.)
- Try use-cases where multiple videos are opened/closed.
- Monitor whether graphic memory usage **drops** after video stops.

---

## 🧠 Summary Table

| Mode | Requires HAL Update? | Memory Saved | How It Works |
|---|---|---|---|
| Composer HAL 3.2 | ✅ Yes | 🔥 Full | HAL clears specific buffer slots |
| Backward-Compatible Mode | ❌ No | ⚠️ Partial | Uses fake 1x1 buffers to overwrite |

---

Would you like me to **draw this as a diagram** or walk through a **real example with video playback and freeing buffers**?

at what stages gpu is used for rendering?

Great question! Let's understand **when and where the GPU is used during rendering in Android**, in **simple terms** and in **stages**.

---

# 🔄 Rendering Pipeline Overview

When an Android app draws something (like a button, image, video), this is the typical flow:

```css
App → Android UI Toolkit (View/Canvas) → Surface → SurfaceFlinger → Display
```

The **GPU (Graphics Processing Unit)** gets involved at **several key stages**.

---

# 🎯 Where GPU is Used

### ✅ 1. App UI Rendering (Skia / HWUI / OpenGL / Vulkan)

- The app uses **Android UI Toolkit**, which uses **Skia (2D rendering engine)**.
- On modern Android, this rendering is **GPU-accelerated** via **OpenGL ES** or **Vulkan**.
- Output is drawn into a **GPU texture** in a **Surface (BufferQueue)**.

🧠 Example:

> You scroll a list — GPU renders the updated list onto a buffer.

---

### ✅ 2. Video Decoding (Optional)

- **Video decoder** can use **hardware acceleration (via GPU or video DSP)**.
- If video is decoded to a GPU-backed buffer (e.g. via `SurfaceTexture`), **GPU may handle color conversion** or composition.

---

### ✅ 3. SurfaceFlinger Composition

- **SurfaceFlinger** gathers surfaces from multiple apps/windows (home, keyboard, video player).
- It **composites** these layers into **one final image**.

Depending on your device:

- GPU does this if:
  - No **hardware composer (HWC)** support.
  - HWC can't handle certain effects (e.g., transparency, transformations).
- Else, **hardware composer (HWC)** takes over and **bypasses GPU**.

🧠 Example:

> Video + Chat Bubble + Navigation Bar → GPU may blend them together.

---

### ✅ 4. Post-Processing (Optional)

- Some devices use GPU to apply **color correction**, **shaders**, or **effects** before displaying.
- This might happen in SurfaceFlinger or hardware display pipeline.

---

## 🔍 Summary Table

| Stage | Is GPU Used? | Notes |
|---|---|---|
| App UI Rendering | ✅ Yes | Skia uses GPU via OpenGL/Vulkan |
| Canvas / View rendering | ✅ Yes | Backed by GPU textures |
| Video decode (SurfaceTexture) | ✅ Sometimes | Depends on codec, buffer type |
| SurfaceFlinger Composition | ✅ Sometimes | Falls back to GPU if HWC can't |
| Display Output | ❌ No | Handled by display controller |

## 📌 Important GPU APIs

| API | Purpose |
|---|---|
| OpenGL ES | Traditional GPU rendering |
| Vulkan | Newer, lower-overhead API |
| RenderScript* | Deprecated, used for compute |
| HWUI | Android UI rendering engine |
| SurfaceTexture | GPU-friendly texture interface |

Would you like a **diagram showing each step**, or an **example walkthrough**, like scrolling a webpage or playing a YouTube video?

Layers and displays

bookmark_border
Layers and displays are two primitives that represent composition work and interactions with the display hardware.

Layers
A layer is the most important unit of composition. A layer is a combination of a surface and an instance of SurfaceControl. Each layer has a set of properties that define how it interacts with other layers. Layer properties are described in the following table:

Property	Description
Positional	Defines where the layer appears on its display. Includes information such as the positions of a layer's edges and its Z order relative to other layers (whether it should be in front of or behind other layers).
Content	Defines how content displayed on the layer should be presented within the bounds defined by the positional properties. Includes information such as crop (to expand a portion of the content to fill the bounds of the layer) and transform (to show rotated or flipped content).
Composition	Defines how the layer should be composited with other layers. Includes information such as blending mode and a layer-wide alpha value for alpha compositing.

Optimization Provides information not strictly necessary to correctly composite the layer, but that can be used by the Hardware Composer (HWC) device to optimize how it performs composition. Includes information such as the visible region of the layer and which portion of the layer has been updated since the previous frame.
Displays
A display is another important unit of composition. A system can have multiple displays and displays can be added or removed during normal system operations. Displays are added or removed at the request of the HWC or at the request of the framework. The HWC device requests displays be added or removed when an external display is connected or disconnected from the device, which is called hotplugging. Clients request virtual displays, whose contents are rendered into an off-screen buffer instead of to a physical display.

Virtual displays
SurfaceFlinger supports an internal display (built into the phone or tablet), external displays (such as a television connected through HDMI), and one or more virtual displays that make composited output available within the system. Virtual displays can be used to record the screen or send the screen over a network. Frames generated for a virtual display are written to a BufferQueue.

Virtual displays may share the same set of layers as the main display (the layer stack) or have their own set. There's no VSync for a virtual display, so the VSync for the internal display triggers composition for all displays.

On HWC implementations that support them, virtual displays can be composited with OpenGL ES (GLES), HWC, or both GLES and HWC. On nonsupporting implementations, virtual displays are always composited using GLES.

Case study: screenrecord
The screenrecord command allows the user to record everything that appears on the screen as an MP4 file on disk. To implement this, the system receives composited frames from SurfaceFlinger, writes them to the video encoder, and then writes the encoded video data to a file. The video codecs are managed by a separate process (mediaserver), so large graphics buffers have to move around the system. To make it more challenging, the goal is to record 60 fps video at full resolution. The key to making this work efficiently is BufferQueue.

The MediaCodec class allows an app to provide data as raw bytes in buffers, or through a surface. When screenrecord requests access to a video encoder, the mediaserver process creates a BufferQueue, connects itself to the consumer side, then passes the producer side back to screenrecord as a surface.

The screenrecord utility then asks SurfaceFlinger to create a virtual display that mirrors the main display (that is, it has all of the same layers), and directs it to send output to the surface that came from the mediaserver process. In this case, SurfaceFlinger is the producer of buffers rather than the consumer.

After the configuration is complete, screenrecord triggers when the encoded data appears. As

> apps draw, their buffers travel to SurfaceFlinger, which composites them into a single buffer that's sent directly to the video encoder in the mediaserver process. The full frames are never seen by the screenrecord process. Internally, the mediaserver process has its own way of moving buffers around that also passes data by handle, minimizing overhead.
>
> Case study: simulate secondary displays
> The WindowManager can ask SurfaceFlinger to create a visible layer for which SurfaceFlinger acts as the BufferQueue consumer. It's also possible to ask SurfaceFlinger to create a virtual display, for which SurfaceFlinger acts as the BufferQueue producer.
>
> If you connect a virtual display to a visible layer, a closed loop is created where the composited screen appears in a window. That window is now part of the composited output, so on the next refresh the composited image inside the window shows the window contents as well. To see this in action, enable Developer options in Settings, select Simulate secondary displays, and enable a window. To see secondary displays in action, use screenrecord to capture the act of enabling the display then play it back frame by frame.

Great, you've shared detailed information on **Layers and Displays** in Android's graphics system, especially from the perspective of **SurfaceFlinger** and **Hardware Composer (HWC)**. Let's now directly answer your earlier question:

---

## 🔧 At what stages is the GPU used for rendering?

In Android's rendering pipeline, **the GPU is used at specific stages depending on the composition strategy** selected by SurfaceFlinger and the capabilities of the Hardware Composer (HWC). Here's a breakdown:

---

### ✅ 1. App Rendering (GPU used):

- **When an app draws its UI**, it uses Android's rendering APIs (`Canvas`, `OpenGL`, `Vulkan`, `RenderScript`, etc.).
- Most modern apps render via **Skia (CPU/GPU)** or **OpenGL ES/Vulkan (GPU)**.
- The **GPU is used here to rasterize** and draw the contents of a `Surface` (backed by a `GraphicBuffer`) into memory.

👉 **Example:**

- An app draws a button or image → GPU rasterizes the content → Frame is placed into a buffer in a `Surface`.

---

### ✅ 2. SurfaceFlinger Composition (GPU sometimes used):

When all app layers are rendered and queued via `BufferQueue`:

**SurfaceFlinger has two options for composing layers:**

---

### ⚙️ A. GPU Composition (GLES):

- If **HWC cannot handle all layers**, or **additional effects like rotation, blending, color conversion** are needed, SurfaceFlinger uses **OpenGL ES (GPU)** to do the final composition.

**GPU used for:**

- **Blending layers**
- **Transforming, scaling**
- **Rendering into framebuffer (FBO)**
- **Handling virtual displays on unsupported HWC**

✅ **This happens in:**

- Virtual display rendering (e.g., screenrecord, cast)
- Complex layer blending (semi-transparent layers)
- Devices without advanced HWC features

---

### ⚙️ B. Hardware Composer (HWC) Composition:

- **No GPU used.**
- SurfaceFlinger delegates composition to HWC, which tells the **Display Engine (not GPU)** how to scanout layers directly from memory.
- This is more power-efficient and faster.

🧠 HWC can:

- Do hardware overlay composition
- Skip GPU if transformations are minimal

---

### ✅ 3. Virtual Display / Screenrecord (GPU used depending on HWC support):

- If HWC does **not support virtual display**, SurfaceFlinger **uses GPU (OpenGL ES)** to composite to an offscreen buffer (FBO).
- This buffer is then passed to the video encoder.

---

### ✅ 4. Final Display Output (GPU not always involved):

- If composition was done by GPU (GLES), the GPU has already written the final frame into a buffer.
- This buffer is sent to the **Framebuffer HAL or DRM**, depending on the platform.
- If composed by HWC, then **no GPU** is used at this stage either.

---

## Summary Table:

| Stage | GPU Used? | Description |
|---|---|---|
| App renders UI | ✅ Yes | UI is rendered using OpenGL/Skia/Vulkan |
| SurfaceFlinger composition | ✅ Sometimes | Uses GLES if HWC can't do it |
| HWC composition | ❌ No | Composition done by Display Engine |
| Virtual display composition | ✅ Usually | GPU used if HWC doesn't support |
| Final display | ❌ Usually not | Framebuffer or display engine handles |

---

## 🧠 Bonus Tip: How SurfaceFlinger Decides Between GPU and HWC

- SurfaceFlinger queries HWC for which layers it can handle (`prepare()`).
- Based on this, it may assign some layers to HWC, and others to GPU.
- This hybrid strategy is called **client composition (GLES) + device composition (HWC)**.

Would you like a visual diagram or code-level explanation of this flow?

# Android graphics HAL-2

Layers and displays

bookmark_border
Layers and displays are two primitives that represent composition work and interactions with the display hardware.

Layers
A layer is the most important unit of composition. A layer is a combination of a surface and an instance of SurfaceControl. Each layer has a set of properties that define how it interacts with other layers. Layer properties are described in the following table:

Property Description
Positional Defines where the layer appears on its display. Includes information such as the positions of a layer's edges and its Z order relative to other layers (whether it should be in front of or behind other layers).
Content Defines how content displayed on the layer should be presented within the bounds defined by the positional properties. Includes information such as crop (to expand a portion of the content to fill the bounds of the layer) and transform (to show rotated or flipped content).
Composition Defines how the layer should be composited with other layers. Includes information such as blending mode and a layer-wide alpha value for alpha compositing.
Optimization Provides information not strictly necessary to correctly composite the layer, but that can be used by the Hardware Composer (HWC) device to optimize how it performs composition. Includes information such as the visible region of the layer and which portion of the layer has been updated since the previous frame.
Displays
A display is another important unit of composition. A system can have multiple displays and displays can be added or removed during normal system operations. Displays are added or removed at the request of the HWC or at the request of the framework. The HWC device requests displays be added or removed when an external display is connected or disconnected from the device, which is called hotplugging. Clients request virtual displays, whose contents are rendered into an off-screen buffer instead of to a physical display.

Virtual displays
SurfaceFlinger supports an internal display (built into the phone or tablet), external displays (such as a television connected through HDMI), and one or more virtual displays that make composited output available within the system. Virtual displays can be used to record the screen or send the screen over a network. Frames generated for a virtual display are written to a BufferQueue.

Virtual displays may share the same set of layers as the main display (the layer stack) or have their own set. There's no VSync for a virtual display, so the VSync for the internal display triggers composition for all displays.

On HWC implementations that support them, virtual displays can be composited with OpenGL ES (GLES), HWC, or both GLES and HWC. On nonsupporting implementations, virtual displays are always composited using GLES.

Case study: screenrecord
The screenrecord command allows the user to record everything that appears on the screen as an MP4 file on disk. To implement this, the system receives composited frames from SurfaceFlinger, writes them to the video encoder, and then writes the encoded video data to a file. The video codecs are managed by a separate process (mediaserver), so large graphics buffers have to move around the system. To make it more challenging, the goal is to record 60 fps video at full resolution. The key to making this work efficiently is BufferQueue.

The MediaCodec class allows an app to provide data as raw bytes in buffers, or through a surface. When screenrecord requests access to a video encoder, the mediaserver process creates a BufferQueue, connects itself to the consumer side, then passes the producer side back to screenrecord as a surface.

The screenrecord utility then asks SurfaceFlinger to create a virtual display that mirrors the main display (that is, it has all of the same layers), and directs it to send output to the surface that came from the mediaserver process. In this case, SurfaceFlinger is the producer of buffers rather than the consumer.

After the configuration is complete, screenrecord triggers when the encoded data appears. As apps draw, their buffers travel to SurfaceFlinger, which composites them into a single buffer that's sent directly to the video encoder in the mediaserver process. The full frames are never seen by the screenrecord process. Internally, the mediaserver process has its own way of moving buffers around that also passes data by handle, minimizing overhead.

Case study: simulate secondary displays
The WindowManager can ask SurfaceFlinger to create a visible layer for which SurfaceFlinger acts as the BufferQueue consumer. It's also possible to ask SurfaceFlinger to create a virtual display, for which SurfaceFlinger acts as the BufferQueue producer.

If you connect a virtual display to a visible layer, a closed loop is created where the composited screen appears in a window. That window is now part of the composited output, so on the next refresh the composited image inside the window shows the window contents as well. To see this in action, enable Developer options in Settings, select Simulate secondary displays, and enable a window. To see secondary displays in action, use screenrecord to capture the act of enabling the display then play it back frame by frame.

This explanation gives a clear overview of **layers** and **displays** in Android's graphics system — primarily how **SurfaceFlinger** and **Hardware Composer (HWC)** collaborate to compose what you see on-screen. Here's a breakdown to simplify the concepts and clarify their roles:

---

# 🔷 1. Layers: The Building Blocks of the Screen

A **layer** represents a single visual element that needs to be composed (drawn) on the screen — like a window, status bar, video frame, or app surface.

Each **layer = Surface + SurfaceControl**.

### 📌 Layer Properties

| Property | What it controls |
|---|---|
| **Positional** | Where to draw the layer on screen (position, size, Z-order) |
| **Content** | How to draw the content (e.g. crop, scale, rotate, flip) |
| **Composition** | How to blend with other layers (e.g. transparency, alpha blending) |
| **Optimization** | Hints for performance (e.g. which regions changed, visible area, etc.) |

➡️ These properties let SurfaceFlinger and HWC know how to **blend and position layers** during each frame of rendering.

---

# 🔷 2. Displays: Where Layers are Shown

A **display** is a target for showing the composited layers — this could be:

- **Internal display** (main phone/tablet screen)
- **External display** (e.g., HDMI monitor)
- **Virtual display** (used for mirroring/recording/sharing the screen)

### 📌 Hotplugging

When a display is physically connected or disconnected (e.g., plugging into HDMI), this is called **hotplugging**. HWC notifies the system to add or remove displays accordingly.

---

# 🔷 3. Virtual Displays: Not Physically Present

A **virtual display** is a display backed by an **off-screen buffer** (not a physical screen). They're used for:

- **Screen recording** (e.g., `screenrecord`)
- **Screen casting**

- **Simulated displays** for testing (e.g., “Simulate Secondary Display” in Developer Options)

### 🧠 Fun fact:

There's **no VSync** for virtual displays — they depend on the main internal display’s VSync to trigger composition.

---

## 🔷 4. How Screen Recording Works (Case Study)

### Flow of `screenrecord`:

1. `screenrecord` asks for a video encoder (MediaCodec).
2. **Media server** creates a `BufferQueue` and gives a **producer surface** to `screenrecord`.
3. `screenrecord` creates a **virtual display** that mirrors the main display, and routes its output to that surface.
4. **SurfaceFlinger** becomes the **producer** — generating composited frames.
5. These frames go **directly to the video encoder** (not via `screenrecord`), minimizing overhead.
6. Encoded frames are written to disk as MP4.

➡️ Buffers are passed by **handle**, not copied — efficient zero-copy design.

---

## 🔷 5. Simulate Secondary Display (Case Study)

Used for testing multi-display behavior without real hardware.

### How it works:

- Developer options → Simulate secondary displays.
- System creates a **virtual display**.
- SurfaceFlinger sends output to a **visible layer** (SurfaceFlinger consumes it).
- You get a recursive/composited window — sort of like a “screen inside screen” effect.

---

## 🔄 Summary

| Term | Role |
|---|---|
| **Layer** | Represents a single compositable unit (app window, surface, etc.) |
| **Display** | Output destination for layers (physical screen or virtual buffer) |
| **SurfaceFlinger** | System compositor — takes all layers, composes them to framebuffers |

| Term | Role |
|---|---|
| **HWC** | Hardware Composer — offloads composition to display hardware |
| **Virtual Display** | Software-only output (recording, casting, simulated displays) |
| **BufferQueue** | Producer-consumer queue for buffer sharing (zero-copy) |

---

Would you like a **diagram** to visualize the composition and display pipeline?

↳

**How Screen Recording Works (Case Study)** ---------------------------------------- ### Flow of `screenrecord`: 1. `screenrecord` asks for a video encoder (MediaCodec). 2. **Media server** creates a `BufferQueue` and gives a **producer surface** to `screenrecord`. 3. `screenrecord` creates a **virtual display** that mirrors the main display, and routes its output to that surface. 4. **SurfaceFlinger** becomes the **producer** — generating composited frames. 5. These frames go **directly to the video encoder** (not via `screenrecord`), minimizing overhead. 6. Encoded frames are written to disk as MP4. ➡️ Buffers are passed by **handle**, not copied — efficient zero-copy design.

explain in detail in simple words

Let's break down **how screen recording works on Android**, step-by-step, in **very simple terms**, like a story:

---

## 🎥 Imagine this:

You want to **record your Android screen** — everything you see: apps, games, videos — and save it as a video file. The tool you use is the command:

```nginx
screenrecord
```

So, what happens **under the hood** when you run it?

---

## 🧩 Step-by-step, in simple words:

---

### ✅ Step 1: screenrecord asks for a video recorder

- The screenrecord tool first asks Android:

➤ "Hey, I need a video recorder to turn screen images into a video file."

- Android uses a special component for this called **MediaCodec**, managed by a system process called **mediaserver**.

---

### ✅ Step 2: mediaserver sets up a buffer

- The **mediaserver** prepares a place where screen images will be sent — this is a **BufferQueue**.

  Think of this like a mailbox: one side puts images in, the other side picks them up.

- **mediaserver** keeps the **reader** side for itself (called the "consumer").
- Then it gives the **writer** side (called the "producer") to **`screenrecord`**. But instead of a mailbox, Android calls this a **Surface**.

---

### ✅ Step 3: screenrecord creates a fake display

- Now, **`screenrecord`** tells **SurfaceFlinger** (the screen composer):

  ➤ "Please create a **virtual display** — one that acts like a screen, but isn't real."

- This virtual display is like a mirror of your phone screen. It looks at whatever is shown on your real screen.
- Then it says:

  ➤ "Send everything you draw on this virtual display into this Surface (from mediaserver)."

---

### ✅ Step 4: SurfaceFlinger draws screen content

- Now, as your apps draw things (like YouTube, Chrome, etc.), their images go to **SurfaceFlinger**.
- SurfaceFlinger collects all those app layers, **composites** them (like stacking transparent slides), and creates the final screen image.
- But now it sends this image to the **virtual display** instead of your real screen.
- So in this case, **SurfaceFlinger becomes the writer** — it puts the final image into the BufferQueue (via the Surface).

---

### ✅ Step 5: mediaserver encodes it to video

- Every time a new frame is ready, **mediaserver** grabs it from the BufferQueue.
- Then it uses the video encoder (**MediaCodec**) to turn the frame into compressed video data (like H.264).

---

### ✅ Step 6: screenrecord saves the video

- Now that mediaserver has the compressed video, `screenrecord` simply saves that data as a **.mp4** file on disk.

---

## 🚀 Why is this efficient?

- **Zero copy**: Images (buffers) are not moved around or copied from app to app. Instead, they are just **passed by handle** — like saying, "Here's where the image is, go read it directly."

  🔁 This saves time and memory.

---

## 🖼️ Summary with roles:

| Component | Role |
| --- | --- |
| `screenrecord` | Starts the process, requests screen capture and saves video |
| **mediaserver** | Sets up the encoder and BufferQueue |
| **BufferQueue** | Connects SurfaceFlinger (producer) to the video encoder (consumer) |
| **SurfaceFlinger** | Composites screen layers and writes final images to the virtual display |
| **MediaCodec** | Encodes images into video |

---

## 💡 Simple Analogy:

Think of screenrecording like setting up a **security camera that watches your phone screen**. The camera (mediaserver) says, "Give me the video feed." SurfaceFlinger sets up a fake monitor (virtual display), mirrors everything it does to the camera's feed, and the camera writes it to a video file.

---

Let me know if you'd like this explained with a **visual diagram**, or if you want to understand how this ties into GPU/HWC!

VSync

bookmark_border
The VSync signal synchronizes the display pipeline. The display pipeline consists of app rendering, SurfaceFlinger composition, and the Hardware Composer (HWC) presenting images on the display. VSync synchronizes the time apps wake up to start rendering, the time SurfaceFlinger wakes up to composite the screen, and the display refresh cycle. This synchronization eliminates stutter and improves the visual performance of graphics.

The HWC generates VSync events and sends the events to SurfaceFlinger through the callback:

typedef void (*HWC2_PFN_VSYNC)(hwc2_callback_data_t callbackData,
    hwc2_display_t display, int64_t timestamp);
Note: The HWC triggers hwc2_callback_data_t from a thread of HAL_PRIORITY_URGENT_DISPLAY with as little latency as possible, typically less than 0.5 ms.
SurfaceFlinger controls whether or not the HWC generates VSync events by calling to setVsyncEnabled. SurfaceFlinger enables setVsyncEnabled to generate VSync events so it can synchronize with the refresh cycle of the display. When SurfaceFlinger is synchronized to the display refresh cycle, SurfaceFlinger disables setVsyncEnabled to stop the HWC from generating VSync events. If SurfaceFlinger detects a difference between the actual VSync and the VSync it previously established SurfaceFlinger re-enables VSync event generation.

VSync offsets
The sync app and SurfaceFlinger render loops to the hardware VSync. On a VSync event, the display begins showing frame N while SurfaceFlinger begins compositing windows for frame N+1. The app handles pending input and generates frame N+2.

Synchronizing with VSync delivers consistent latency. It reduces errors in apps and SurfaceFlinger and minimizes displays drifting in and out of phase with each other. This assumes app and SurfaceFlinger per-frame times don't vary widely. The latency is at least two frames.

To remedy this, you can employ VSync offsets to reduce the input-to-display latency by making app and composition signal relative to hardware VSync. This is possible because app plus composition usually takes less than 33 ms.

The result of a VSync offset is three signals with same period and offset phase:

HW_VSYNC_0 — Display begins showing next frame.
VSYNC — App reads input and generates next frame.
SF_VSYNC — SurfaceFlinger begins compositing for next frame.
With VSync offsets, SurfaceFlinger receives the buffer and composites the frame while the app simultaneously processes the input and renders the frame.

Note: VSync offsets reduce the time available for app and composition, providing a greater chance for error.
DispSync
DispSync maintains a model of the periodic hardware-based VSync events of a display and uses that model to execute callbacks at specific phase offsets from the hardware VSync events.

DispSync is a software phase-lock loop (PLL) that generates the VSYNC and SF_VSYNC signals used by Choreographer and SurfaceFlinger, even if not offset from hardware VSync.

The DispSync flow is illustrated in the following figure:

DispSync flow
Figure 1. DispSync flow.

DispSync has the following qualities:

Reference — HW_VSYNC_0.
Output — VSYNC and SF_VSYNC.
Feedback — Retire fence signal timestamps from the HWC.
VSync retire offsets
The signal timestamp of retire fences must match HW VSync, even on devices that don't use the offset phase. Otherwise, errors appear to be more severe than they are. Smart panels often have a delta where the retire fence is the end of direct memory access (DMA) to display memory, but the actual display switch and HW VSync is some time later.

PRESENT_TIME_OFFSET_FROM_VSYNC_NS is set in the device's BoardConfig.mk makefile. It's based on the display controller and panel characteristics. The time from the retire fence timestamp to the HW VSync signal is measured in nanoseconds.

VSYNC and SF_VSYNC offsets
VSYNC_EVENT_PHASE_OFFSET_NS and SF_VSYNC_EVENT_PHASE_OFFSET_NS are set conservatively based on high-load use cases, such as partial GPU composition during window transition or Chrome scrolling through a web page containing animations. These offsets allow for long app render time and long GPU composition time.

More than a millisecond or two of latency is noticeable. To minimize latency without significantly increasing error counts, integrate thorough automated error testing.

Note: VSYNC and SF_VSYNC offsets are also configured in the device's BoardConfig.mk file. Both settings are offset in nanoseconds after HW_VSYNC_0, default to zero (if not set), and can be negative.

Let’s simplify and explain the concept of **VSync** and how it works in Android, step-by-step, like a story — including **why it's needed, how it's generated, and what offsets mean**.

---

# 🌟 What is VSync?

**VSync** (Vertical Synchronization) is a signal used to **coordinate the timing** of:

- **Apps** rendering graphics
- **SurfaceFlinger** compositing layers
- **Hardware Composer (HWC)** showing the final frame on screen

Think of VSync like a **metronome** or **timer** that keeps everyone in sync. This prevents tearing, stuttering, and weird glitches when drawing images on the screen.

---

## 🎯 Why is VSync important?

Without synchronization:

- An app may draw a new frame **while** the screen is still updating → causes **screen tearing**
- SurfaceFlinger may miss a new frame → causes **lag or stutter**

So Android synchronizes everyone with the VSync signal to achieve:
✅ Smooth animation
✅ Consistent frame delivery
✅ Reduced visual artifacts

---

## 🧠 Who generates VSync?

### ➤ HWC (Hardware Composer)

HWC is part of Android's HAL layer and knows when the screen is refreshing. It generates **VSync events** using this callback:

```cpp
typedef void (*HWC2_PFN_VSYNC)(
    hwc2_callback_data_t callbackData,
    hwc2_display_t display,
    int64_t timestamp);
```

It sends the timestamp to **SurfaceFlinger**, which uses it to plan the next frame.

---

## ⚙️ Who enables/disables VSync?

- **SurfaceFlinger controls** whether HWC should send VSync events using:

```cpp
setVsyncEnabled(true/false)
```

- Why disable it? To save CPU/GPU if the screen is idle (nothing is changing).

---

## 🔁 VSync Cycle: What happens on each VSync?

Imagine you are watching frame **N** now.

| Time | Component | What it does |
|---|---|---|
| Now | Display | Shows **frame N** |
| Now | SurfaceFlinger | Starts compositing **frame N+1** |
| Now | App | Prepares to draw **frame N+2** |

This happens on **every VSync**, like a factory assembly line.

---

# 🕐 Problem: Latency

All of this takes time. Even if the display refreshes at 60Hz (every ~16.67ms), the app and SurfaceFlinger work **1–2 frames ahead**, causing **visual delay**.

---

# ✅ Solution: VSync Offsets

## What is a VSync offset?

Instead of triggering everything exactly at HW_VSYNC_0, we offset the timing slightly:

| Signal | Meaning |
|---|---|
| **HW_VSYNC_0** | Hardware says: "Start showing next frame" |
| **VSYNC** | App starts rendering |
| **SF_VSYNC** | SurfaceFlinger starts compositing |

They are like 3 bells ringing at slightly different times, but in sync with the main clock.

This way:

- App rendering and SF compositing **can overlap**
- Total latency is **reduced**

---

# ⚙️ Who manages these offsets?

## ➤ DispSync

This is a **software module** in SurfaceFlinger. It:

- **Models** hardware VSync using timestamps
- Creates **offset signals** (VSYNC and SF_VSYNC)
- Ensures they stay **in phase** even if HWC drifts a bit

It's like a **software metronome** that syncs apps and SurfaceFlinger with display hardware.

## 📏 Where are offsets defined?

These values are set in the **BoardConfig.mk** file of your Android device:

| Property | Description |
|---|---|
| `PRESENT_TIME_OFFSET_FROM_VSYNC_NS` | Delay between retire fence and actual VSync |
| `VSYNC_EVENT_PHASE_OFFSET_NS` | When app should wake up (relative to HW_VSYNC) |
| `SF_VSYNC_EVENT_PHASE_OFFSET_NS` | When SurfaceFlinger should wake up (relative to VSync) |

These are in **nanoseconds**, can be **positive or negative**, and are tuned for performance.

## 🧪 Example:

Let's say:

- HW refresh is every 16.67 ms (60 FPS)
- `VSYNC_EVENT_PHASE_OFFSET_NS = -2ms`
- `SF_VSYNC_EVENT_PHASE_OFFSET_NS = 0ms`

This means:

- **App wakes up 2ms before hardware VSync**
- **SurfaceFlinger wakes up at VSync**

Now SurfaceFlinger and the app work in **parallel**, and the final frame shows up faster.

## 🧠 Retire Fence?

A **retire fence** is a signal from HWC that says:

> "I've finished reading this buffer for display."

This helps SurfaceFlinger **recycle** buffers safely. The fence timestamp must match HW_VSYNC time closely; if not, it causes **timing bugs** and visible glitches.

## 🧩 Final Summary

| Concept | Meaning |
|---|---|
| **VSync** | Signal to sync screen updates |

| Concept | Meaning |
|---|---|
| **HW_VSYNC_0** | Real hardware refresh signal |
| **VSYNC** | When the app should render |
| **SF_VSYNC** | When SurfaceFlinger should composite |
| **DispSync** | Software that maintains these signals and phase offsets |
| **Retire Fence** | Signal from HWC: buffer is done displaying |
| **Offsets** | Delays (in ns) to optimize rendering & reduce latency |
| **BoardConfig.mk** | Where offsets are configured |

---

Let me know if you'd like a **timeline diagram** or animation-style visual to explain these phases better!

↳

VSync offset
explain in sinmple words wiht example

Sure! Let's explain **VSync offset** in **very simple words** — with an example — so it's easy to understand.

---

## 🎯 First, what is VSync?

When your phone's screen updates (like 60 times per second), that's called **VSync** – short for **Vertical Synchronization**.

So at **every 16.67 milliseconds**, the display says:

> "Okay, I'm ready to show the next frame."

We call that moment **HW_VSYNC_0**.

---

## 👷 Who works at each VSync?

There are 3 workers involved in each frame of drawing:

| Worker | What they do |
|---|---|
| **App** | Draws new content (e.g., game, video) |
| **SurfaceFlinger** | Combines layers from apps |
| **Display (HWC)** | Shows the final image |

---

# 💡 The Problem

If **everyone waits** until HW_VSYNC to start working, it takes too long — maybe you'll miss the next screen refresh!

We want the app and SurfaceFlinger to **start a little earlier**, so the work is ready **just in time**.

---

# ✅ Solution: VSync offsets

Think of the screen update (VSync) as a **school bell** ringing every 16.67 ms.

### Without offset:

- App starts drawing **when bell rings**
- SurfaceFlinger starts **when bell rings**
- ➡️ Too late! Frame might be **missed**

### With offset:

- App starts **just before** the bell
- SurfaceFlinger starts **a bit after** the app
- So everything is **ready by the next bell**

---

# 🧪 Real-world example

Let’s say:

- Display refreshes: every **16.67 ms** (60 Hz)
- Offsets:
  - App starts drawing **2 ms before** VSync
  - SurfaceFlinger starts **right at** VSync

So if the **bell (HW_VSYNC_0)** rings at time = 100 ms:

| Time | Event |
|---|---|
| 98 ms | App starts drawing next frame |
| 100 ms | SurfaceFlinger starts combining layers |
| 116.67 ms | Display shows the final image |

This way, by the time the screen is ready, the frame is already drawn and composed — **no lag, no tearing**.

## 🧭 Where is this configured?

These offsets are set in Android's **BoardConfig.mk** file:

- `VSYNC_EVENT_PHASE_OFFSET_NS` → app timing
- `SF_VSYNC_EVENT_PHASE_OFFSET_NS` → SurfaceFlinger timing

The values are in **nanoseconds** (1ms = 1,000,000 ns).

---

## 📦 Summary

| Concept | Simple meaning |
|---|---|
| **VSync** | A signal from the screen: "Time to show next frame" |
| **Offset** | Start work **before** the VSync to finish just in time |
| **Why?** | To reduce **latency** and avoid missing the next refresh |
| **Example** | App draws at 98ms, SurfaceFlinger at 100ms, screen shows at 116ms |

---

Would you like a **drawing/timeline** to help make it visual?

Frame pacing

bookmark_border
The Android Frame Pacing library, also known as Swappy, is part of the Android Game SDK. It helps OpenGL and Vulkan games achieve smooth rendering and correct frame pacing on Android.

Frame pacing is the synchronization of a game's logic and rendering loop with an OS's display subsystem and underlying display hardware. The Android display subsystem was designed to avoid certain visual artifacts, such as tearing. The display subsystem avoids tearing by doing the following:

Buffering past frames internally
Detecting late frame submissions
Continuing to display the current frame when a late frame is detected
Inconsistent frame display times are caused by a game's render loop running at a different rate than what the native display hardware supports. Problems arise when a game's render loop runs too slowly for the underlying display hardware, leading to inconsistent display times. For example, when a game running at 30 fps attempts to render on a device that natively supports 60 fps, the game's render loop causes a repeated frame to remain on the screen for an extra 16

ms. This type of disconnect creates substantial inconsistencies in frame times such as 33 ms, 16 ms, and 49 ms. Overly complex scenes further compound this problem because they cause missed frames to occur.

The Frame Pacing library performs these tasks:

Compensates for stuttering due to short game frames.
Adds presentation timestamps so that frames are presented on time, not early.
Uses presentation timestamp extensions EGL_ANDROID_presentation_time and VK_GOOGLE_display_timing.
Uses sync fences for long frames that lead to stuttering and latency.
Injects waits into the app. These allow the display pipeline to catch up, rather than allowing back pressure to build up.
Uses sync fences (EGL_KHR_fence_sync and VkFence).
Chooses a refresh rate to provide flexibility and a smooth presentation, if your device supports multiple refresh rates.
Provides stats for debugging and profiling using frame stats.
To learn how to configure the library to operate in different modes according to what you need, see Supported operating modes.

To implement using OpenGL renderer or Vulkan renderer, see:

Integrate Android Frame Pacing into your OpenGL renderer
Integrate Android Frame Pacing into your Vulkan renderer
To read more, see Frame Pacing library.

Frames per second throttling intervention
Frames per second (FPS) throttling intervention enables games to pace at an appropriate FPS using only platform side changes and without requiring any action on the developers' part.

The implementation of the FPS throttling intervention uses the following components.

GameManagerService
The GameManagerService component maintains all per-user and per-game information of game mode and game intervention. The FPS information is stored in the GameManagerService with other intervention information, such as the resolution downscaling factor, in a <PACKAGE_NAME, Interventions> mapping for each user profile. The FPS information is accessed when the game mode is changed or the intervention is updated. A UID is unique to each PACKAGE_NAME and user, and can further be translated into a <UID, Frame Rate> pair to send to the SurfaceFlinger.

SurfaceFlinger
SurfaceFlinger component already supports throttling the FPS of an application as long as the frame rate is a divisor of the display refresh rate. In the event of a VSync, the SurfaceFlinger checks the validity of the VSync for the throttled application by verifying whether the VSync timestamp is in phase with the frame rate of the application. If the frame rate is not in phase

> with the VSync, then the SurfaceFlinger holds the frame until the frame rate and VSync are in phase.
>
> The following figure describes the interaction between the GameManagerService and the SurfaceFlinger:
>
> Interaction between the GameManagerService and the SurfaceFlinger
> Figure 1. Interaction between the GameServiceManager and SurfaceFlinger.
>
> The SurfaceFinger maintains a <UID, Frame Rate> pair mapping to set a new frame rate throttling priority. The UID is unique between users and games, so that each user on a single device can have different settings of frame rate on the same game. To throttle the frame rate of a game, the GameServiceManager calls the SurfaceFlinger to override the frame rate for a UID. With this mechanism, the SurfaceFlinger updates the mapping whenever the game mode is changed or the intervention is updated. The SurfaceFlinger handles the FPS change by latching buffers accordingly.
>
> To understand more about FPS throttling, see FPS throttling.

Let's break down **Frame Pacing** on Android — what it is, why it matters, and how Android makes games smoother using **Swappy**, **Frame Pacing library**, and **FPS throttling** — in **simple words with real examples**.

---

## 🎮 What is Frame Pacing?

**Frame pacing** means making sure that your game **draws and shows frames** at regular intervals — like a metronome ticking evenly.

### ✅ Good Frame Pacing:

- 60 FPS = every frame shows for exactly **16.67ms**
- Looks smooth to the user.

### ❌ Bad Frame Pacing:

- Some frames take 16ms, some 33ms, some 49ms → causes **stutter**, even if the average is 30 FPS.

---

## 🚧 The Problem Without Frame Pacing

Imagine your game runs at **30 FPS** (draws 1 frame every 33ms), but the screen refreshes at **60Hz** (every 16.67ms). What happens?

- Frame 1: shown for 33ms (good)

- Frame 2: comes late, so frame 1 is shown again (bad)
- Frame 2: now comes too early (bad again)
- Result: **jumpy animations**, flickering movement.

Now imagine some scenes are **heavy**, so the game drops to 22 FPS or lower... That causes **even worse jitter**.

---

## 🛠️ What does the Frame Pacing Library (Swappy) do?

Swappy is part of the **Android Game SDK**. It helps fix these timing issues **automatically**.

### It does:

| What it does | Why it's helpful |
|---|---|
| Adds **presentation timestamps** | So the display shows the frame at the exact right time |
| Uses **sync fences** | So long frames don't block others or create backlogs |
| Adds **delays/waits** if you're too early | To align with display refresh and avoid stutter |
| Picks best **refresh rate** | If device supports multiple (e.g., 60Hz, 90Hz, 120Hz) |
| Helps you **profile/debug** using stats | See where frames are delayed or skipped |

---

## 💡 Simple Analogy

### 🎬 Without frame pacing:

Imagine a movie where some scenes play fast, and some freeze randomly. Even if it's "30 FPS on average," it looks **choppy**.

### 🎬 With frame pacing (Swappy):

Now every frame is timed exactly to the screen — no skipping, no stuttering — so it feels **smooth and natural**.

---

## 🧩 What about FPS throttling?

Android also allows the **system itself** to control your game's FPS. This is called **FPS throttling**.

You don't need to do anything — Android will:

1. Check game settings (from **GameManagerService**).
2. Decide to run game at 30, 45, or 60 FPS, etc.

3. Ask **SurfaceFlinger** to enforce that frame rate.

---

# 👀 How does FPS throttling work?

### Key system parts:

| Component | What it does |
|---|---|
| **GameManagerService** | Stores info per game: FPS, resolution, user settings |
| **SurfaceFlinger** | Uses that info to make sure game frames follow the chosen FPS |

So if a game is supposed to run at 30 FPS on a 60 Hz screen:

- SurfaceFlinger only accepts **1 frame every 2 VSyncs**.
- If a frame comes early, it **waits** until it's time to show it.
- This keeps the game in **sync** with the display.

---

# 🔁 Example Scenario: Game running at 30 FPS on 60 Hz display

| Time | What happens |
|---|---|
| 0 ms | Game draws frame 1 |
| 16.67 ms | Screen is ready for next frame, but SurfaceFlinger **waits** |
| 33.33 ms | Shows frame 2 from game |
| 50.00 ms | SurfaceFlinger waits again (game is throttled) |
| 66.67 ms | Shows frame 3 |

➡️ This keeps game **smooth**, even at a lower frame rate.

---

# 📦 Summary

| Concept | Simple Explanation |
|---|---|
| **Frame pacing** | Making sure each frame is shown evenly and at the right time |
| **Swappy (Frame Pacing)** | Helps time frames perfectly using timestamps, fences, waits |
| **FPS throttling** | Android system limits your FPS to match display (e.g., 30, 45, 60 FPS) |
| **GameManagerService** | Keeps per-game settings (FPS, resolution, etc.) |
| **SurfaceFlinger** | Enforces those settings, holds or releases frames as needed |

---

Let me know if you'd like:

- A diagram showing this timeline visually

- Example code using Swappy (OpenGL/Vulkan)
- Explanation of how `EGL_ANDROID_presentation_time` or `VK_GOOGLE_display_timing` works in practice

Multiple refresh rate

bookmark_border
Android 11 adds support for devices with multiple refresh rates. There are three main components to this feature:

New HAL APIs introduced in android.hardware.graphics.composer@2.4.
Platform code to parse device configs for different refresh rates and set the desired refresh rate
New SDK and NDK APIs to allow apps to set their desired frame rate
Implementation
Dedicated support for refresh rate switching has been added to android.hardware.graphics.composer@2.4 HAL. We strongly recommend using this version since previous versions of composer HAL have limited support for refresh rate switching.

Config groups
A new attribute CONFIG_GROUP was added to IComposerClient::Attribute that is queryable using the getDisplayAttribute_2_4 API. This attribute allows vendors to group together display configurations. Configurations in the same group allows seamless switching between them in most of the cases. The config group is used by the platform to differentiate which configurations can be switched between them in order to switch the refresh rate and not other attributes for a config.

Consider the following example that demonstrates the benefits of using config groups with a device that supports four display configurations:

1080p@60Hz
1080p@90Hz
1080i@72Hz
1080i@48Hz
Even though the device supports 48Hz, 60Hz, 72Hz, and 90Hz refresh rates the display operates at a different mode and switching from 60Hz to 72Hz changes the display config from 1080p to 1080i, which might not be the desired behavior. This is solved by using config groups. By grouping 60Hz and 90Hz together in one config group and 48Hz and 72Hz in another config group. The platform knows that it can switch between 60Hz and 90Hz and between 48Hz and 72Hz but not between 60Hz and 72Hz as this will result in a config change rather than simply changing the refresh rate.

Composer API updates
getDisplayVsyncPeriod

For better control and predictability when changing refresh rates getDisplayVsyncPeriod has been added. getDisplayVsyncPeriod returns the current refresh rate (in terms of vsync period) the display operates at. This is especially useful while transitioning between refresh rate and the current refresh rate is needed by the platform to decide when to start the next frame.
setActiveConfigWithConstraints
The setActiveConfigWithConstraints method is a new extension to the existing setActiveConfig method and provides more information about the config change. The constraints are given as part of the vsyncPeriodChangeConstraints parameters and contain the following parameters.
desiredTimeNanos
The time in CLOCK_MONOTONIC after which the vsync period may change (that is the vsync period must not change before this time). This is useful when the platform wants to plan ahead for a refresh rate change but it already has some buffers in the queue to present. The platform sets this time accordingly to account for those buffers and make sure that the refresh rate transition will be as smooth as possible.
seamlessRequired
If true, requires that the vsync period change must happen seamlessly without a noticeable visual artifact. This flag is used by the platform when a refresh rate change is needed as a result of a content change (for example, the device is idle and animation starts). This gives the vendor the opportunity to not allow certain configuration changes when they might result in a noticeable visual artifact. If the configs can't be changed seamlessly and seamlessRequired is set to true, the implementation is expected to return SEAMLESS_NOT_POSSIBLE as the return code and call the new onSeamlessPossible callback when the same config change can be done seamlessly.
Upon success the implementation returns a VsyncPeriodChangeTimeline which tells the platform when to expect the refresh rate change to occur. newVsyncAppliedTimeNanos parameters need to be set to the time in CLOCK_MONOTONIC when the new display will start to refresh at the new vsync period. This, together with desiredTimeNanos allows the platform to plan in advance the refresh rate switch and start ticking apps for the new refresh rate in advance. This allows a seamless transition of the refresh rate.

Some implementations require a refresh frame to be sent before the refresh rate can be sent. For that, the HAL has the refreshRequired parameter to indicate that a refresh frame is needed and refreshTimeNanos to indicate the first vsync where a refresh frame needs to be sent after.

onVsyncPeriodTimingChanged [callback]
A new callback that can be called by the HAL to indicate to the platform that some parameter of the timeline changed and the platform needs to adjust its timeline. This callback is expected to be called if for some reason the old timeline was missed because of a long processing time on the HAL or a late refresh frame.
How does the platform decide to change the refresh rate?
The refresh rate selection happens in the following two system services:

DisplayManager
The DisplayManager sets the high level policy around the refresh rate. It sets a default display config, which is the same as the composer HAL config. Additionally, it sets a range of minimum and maximum values for SurfaceFlinger to choose as the refresh rate.

SurfaceFlinger
Determines the refresh rate by setting a config which is in the same config group as the default config and with a refresh rate within the min/max range.
The Display Manager runs through the following steps to determine the policy:

Finds the default config ID by querying the active config from SurfaceFlinger
Restricting the range of minimum and maximum values by iterating over the system conditions
Default refresh rate setting: The default refresh rate value is set in the R.integer.config_defaultRefreshRate config overlay. This value is used to determine the standard device refresh rate for animations and touch interactions.
Peak refresh rate setting: The peak refresh rate value is read from Settings.System.PEAK_REFRESH_RATE. This value is changed in runtime to reflect the current device setting (such as from a menu option). The default value is set in the R.integer.config_defaultPeakRefreshRate config overlay.
Minimum refresh rate setting: The minimum refresh rate value is read from Settings.System.MIN_REFRESH_RATE. This value can be changed in runtime to reflect the current device setting (such as from a menu option). The default value is 0, so there is no default minimum.
Application requested ModeId: Apps can set WindowManager.LayoutParams.preferredDisplayModeId to reflect a preferred config the display should operate at. In most conditions the DisplayManager sets the default config ID accordingly and sets the minimum and maximum refresh rate to match the config's refresh rate.
Battery Saver: The refresh rate is restricted to 60Hz or lower when the device is in low power mode, which is indicated through Settings.Global.LOW_POWER_MODE.
Once DisplayManager sets the policy, SurfaceFlinger sets the refresh rate based on the active layers (layers that queue frame updates). If the owner of the layer sets a frame rate, SurfaceFlinger tries to set the refresh rate to something that is a multiplier of that rate. For example if two active layers set their frame rate to 24 and 60 SurfaceFlinger will pick 120Hz if it is available. If such refresh rate is not available to SurfaceFlinger, it will try to pick the refresh rate which has the minimal error for the frame rate. For more information see the developer documentation on developer.android.com

SurfaceFlinger maintains the following flags to control how the refresh rate is decided:

ro.surface_flinger.use_content_detection_for_refresh_rate: If set the refresh rate is decided based on the active layers, even if a frame rate was not set. SurfaceFlinger maintains a heuristic where it finds the average fps the layer is posting buffers by looking at the presentation timestamp attached to the buffer.
ro.surface_flinger.set_touch_timer_ms: if > 0, the default refresh rate will be used when a user touches the screen for the configured timeout. This heuristic is done to be ready with the default refresh rate for animations.
ro.surface_flinger.set_idle_timer_ms: if > 0, min refresh rate will be used when there are no screen updates for the configured timeout.
ro.surface_flinger.set_display_power_timer_ms: if > 0, the default refresh rate will be used when turning on the display (or when going out of AOD) for the configured timeout.

Frame Rate API
The frame rate API lets apps inform the Android platform of their intended frame rate and is available on apps that target Android 11. To learn more about the frame rate API, check out the developer documentation on developer.android.com.

Developer options

A new developer option has been added to the menu that toggles an overlay on the display with the current refresh rate. The new option is under Settings > System > Developer options > Show refresh rate.

Let's break down **multiple refresh rates on Android** in very **simple words**, with a **real-world example**, so it's easy to understand.

---

## 🖥️ What is a refresh rate?

The **refresh rate** is how many times the screen updates per second.

| Refresh Rate | Meaning |
|---|---|
| 60Hz | Updates 60 times per second |
| 90Hz | Updates 90 times per second |
| 120Hz | Super smooth animations |
| 30Hz | Slower updates (saves power) |

---

## 🎯 Why support multiple refresh rates?

Different apps need different things:

- **Games** like 90Hz or 120Hz → smoother action
- **Videos** like 24Hz or 48Hz → matches movie frame rate
- **Battery saving** needs 30Hz or 60Hz
- **Static screen** (like reading) → 30Hz or lower saves power

So modern Android phones can switch between refresh rates **on the fly**.

---

## 📦 How Android supports multiple refresh rates

Starting from **Android 11**, Android supports:

1. **HAL support** → low-level hardware handling (composer@2.4)

2. **System logic** → Android platform decides when/what to switch
3. **App control** → apps can **ask** for a specific refresh rate

---

# 🧪 Real-World Example

Imagine your phone supports these modes:

- 1080p @ 60Hz
- 1080p @ 90Hz
- 1080i @ 48Hz
- 1080i @ 72Hz

We don't want to switch between 1080p and 1080i — it changes screen resolution. That looks **ugly** and causes **glitches**.

So Android uses **config groups**:

| Config Group | Modes inside |
|---|---|
| Group A | 1080p @ 60Hz, 90Hz |
| Group B | 1080i @ 48Hz, 72Hz |

**Now Android knows** it can **smoothly switch** between 60Hz ↔ 90Hz but **not** 60Hz ↔ 72Hz.

---

# 🔁 How the switch works

## Step-by-step:

1. App or system wants to change refresh rate
2. Android checks if switch is **smooth** (same config group)
3. If yes:
    - Uses **`setActiveConfigWithConstraints()`**
    - Can say:
        - "Switch after 2 frames" (using **`desiredTimeNanos`**)
        - "Must be seamless" (using **`seamlessRequired = true`**)
4. Android waits for the right time
5. Switch happens without flicker or glitch

If a seamless switch is **not possible**, the system **waits** and tries later (using a callback).

---

## ⚙️ Who decides the refresh rate?

| Component | Role |
| --- | --- |
| **DisplayManager** | Sets the **policy**: default, min, max rates |
| **SurfaceFlinger** | Picks best refresh rate based on app needs and current content |

SurfaceFlinger watches **which layers are active** and **how fast they update** (like video or game), and picks the right refresh rate.

---

## 🎮 App and Game Support (Frame Rate API)

Apps can now say:

> "I prefer 30 FPS"
> "I want 60 FPS"
> "Please match my video's 24 FPS"

This is done using `WindowManager.LayoutParams.preferredDisplayModeId`.

Apps targeting Android 11+ can use this to **request** a specific refresh rate. The system **tries** to honor it.

---

## 🔌 Battery saver & user controls

The system changes refresh rate based on:

| Situation | What happens |
| --- | --- |
| **Battery Saver** | Limits refresh rate to save power |
| **Touching screen** | Boosts refresh rate for smooth feel |
| **Idle screen** | Drops to low refresh rate |
| **Developer setting** | Lets devs **see refresh rate live** |

---

## 🧭 Example flags used in SurfaceFlinger

You (as a device maker or developer) can control behavior using these system properties:

| Flag | Meaning |
| --- | --- |
| `ro.surface_flinger.use_content_detection_for_refresh_rate` | Switch based on content (even if app didn't request rate) |
| `ro.surface_flinger.set_touch_timer_ms` | After touch, stay at default rate for X ms |
| `ro.surface_flinger.set_idle_timer_ms` | If screen is idle, drop to lowest refresh rate |

## 👓 Summary (in super simple words)

| Thing | What it does |
|---|---|
| Multiple refresh rate | Screen can switch between 60Hz, 90Hz, 120Hz etc. |
| Config group | Keeps compatible refresh rates together for **smooth switching** |
| DisplayManager | Sets refresh rate policy |
| SurfaceFlinger | Chooses the **best refresh rate** based on active apps |
| Apps (games/video) | Can ask Android for a preferred frame rate |
| Battery/Idle state | System changes refresh rate to **save power** when needed |

---

Let me know if you'd like a **diagram** showing how config groups and switching flow work, or a **sample app** that uses the Frame Rate API!

↳

We don't want to switch between 1080p and 1080i — it changes screen resolution. That looks **ugly** and causes **glitches**. So Android uses **config groups**: | Config Group | Modes inside | | --- | --- | | Group A | 1080p @ 60Hz, 90Hz | | Group B | 1080i @ 48Hz, 72Hz | **Now Android knows** it can **smoothly switch** between 60Hz ↔ 90Hz but **not** 60Hz ↔ 72Hz.
explain in simple words

Sure! Let's break it down in **simple words** with a **real-world example**:

---

### 📺 The Problem: Ugly Glitches When Switching Display Modes

Imagine you're playing a game or watching a video on your phone or TV. The screen can run at different **refresh rates** (like 60Hz, 90Hz, etc.), and different **resolutions** (like 1080p, 1080i).

If the system suddenly switches from **1080p to 1080i**, even if the refresh rate is close, your screen may:

- **Flash**
- **Pause**
- **Look broken for a second**

This is because the **resolution format changed**, not just the speed of the screen.

---

### ✅ The Solution: Config Groups

Android puts similar display modes into **"config groups"**, like this:

| Group | Modes |
|---|---|
| A | 1080p @ 60Hz, 90Hz |
| B | 1080i @ 48Hz, 72Hz |

Each group has **modes with the same resolution**, so:

- **Switching within a group (e.g., 60Hz ↔ 90Hz)** is **smooth** (no glitches)
- **Switching across groups (e.g., 60Hz ↔ 72Hz)** is **not allowed**, because it might **change resolution format** and cause glitches

---

## 🎮 Simple Example:

You're playing a game at:

- **1080p @ 60Hz**

The game wants to go to **faster animation**, so it tries to switch to:

- ✅ **1080p @ 90Hz** → **Allowed** (same group, smooth switch)

But if it tries to switch to:

- ❌ **1080i @ 72Hz** → **Not allowed** (different group, would cause glitch)

---

## 🧠 Why Android Does This

Android uses config groups to:

- **Avoid screen glitches**
- **Keep the user experience smooth**
- **Only switch between refresh rates that are guaranteed safe**

---

Let me know if you want this with a picture or animation-style explanation too!

Adaptive refresh rate

bookmark_border
Starting in Android 15, the adaptive refresh rate (ARR) feature enables the display refresh rate to adapt to the content frame rate, using discrete VSync steps.

The ARR feature offers the following benefits:

Reducing power consumption: By default, ARR enables devices to operate at rates lower than their maximum refresh rates, transitioning to higher rates only when essential for the user experience, which minimizes unnecessary power consumption.

Reducing jank: ARR eliminates the need for mode switching, which is a known cause for jank.

Overview
On non-ARR panels, the display refreshes at a fixed cadence determined by the active display mode.

On ARR panels, the display VSync rate and refresh rate are decoupled, allowing refresh rates to change within one display mode, based on content update cadence. The panels can run at refresh rates that are divisors of the panel's tearing effect (TE). OEMs have the flexibility to implement ARR based on their preferred power trade-offs.

The following figure is a display with vsyncPeriod of 240 Hz and minFrameIntervalNs (maximum refresh rate) of 120 Hz. VSync occurs every 4.16 ms. A frame can be presented at any multiples of VSync after the minFrameIntervalNs from the last frame.

arr-example

Figure 1. Example of ARR.

Implementation
Android 15 supports ARR with new Hardware Composer (HWC) HAL APIs and platform changes. To enable ARR, OEMs must support kernel and system changes on devices running Android 15 and later, and implement version 3 of the android.hardware.graphics.composer3 APIs, as listed in the following sections.

See Pixel's reference implementation of the APIs that support ARR for more information.

Note: The AOSP source code mentions vrr in relation to ARR elements.
DisplayConfiguration.aidl
The DisplayConfiguration.aidl API specifies the display configuration using display attributes, along with the following attributes for ARR:

Optional vrrConfig: If set, ARR is enabled for specific configurations. If set to null, the display mode is set to non-ARR modes such as multiple refresh rate (MRR). With this attribute, a display can be configured as either MRR or ARR, but not both.
vsyncPeriod: The VSync rate of the display. On ARR displays, this value is used to derive the supported discrete refresh rates.

Vendors must set the DisplayConfiguration.vsyncPeriod value for all devices. For non-ARR displays, DisplayConfiguration.vsyncPeriod is the display refresh rate. If a device supports 120

Hz, then this value must be 8.3 ms.

For ARR displays, DisplayConfiguration.vsyncPeriod is the TE signal frequency. If a device has a minFrameIntervalNs of 8.3 ms but the TE is 240 Hz, this value must be 4.16 ms.

VrrConfig.aidl
The VrrConfig.aidl API includes the following attributes:

minFrameIntervalNs: The max refresh rate the display can support.
NotifyExpectedPresentConfig: This is determined by when the display requires advance notice of an upcoming frame.
IComposerClient.notifyExpectedPresent provides a hint for a frame that is likely to be presented, so that the display can adapt its self-refresh period accordingly. frameIntervalNs represents the present cadence that follows after the expectedPresentTime. For example, if notifyExpectedPresent is called with expectedPresentTime N and frameIntervalNs of 16.6 ms, then the next frame is at N + 16.6 ms after the present time N. After the present time N, the frame cadence is 16.6 ms until there are further changes.

IComposerClient.notifyExpectedPresent is called only when DisplayConfiguration.notifyExpectedPresentConfig is set, and if one of the following timing conditions occurs:

Out-of-cadence present time: The expected presentation time of the next frame deviates from the display's regular refresh rate defined by frameIntervalNs.
Timeout exceeded: The time interval between the previous frames is greater than or equal to notifyExpectedPresentConfig.timeoutNs.
DisplayCommand.frameIntervalNs
DisplayCommand.frameIntervalNs provides a hint about the cadence for the upcoming frames in nanoseconds.

Testing
Use onRefreshRateChangedDebug for debugging. This method notifies the client that the refresh rate of the display has changed.

Use the TouchLatency test app for manual testing as shown in Figure 2:

touchlatency-app

Figure 2. TouchLatency test app.

In the test app, use the slider to adjust the render rate to various values of divisor refresh rates of your display's refresh rate. Observe how the frame rate changes with respect to the rate requested.

Sure! Let me explain **Adaptive Refresh Rate (ARR)** in very **simple words**, using **examples**, so it's easy to understand.

---

## 🔁 What is Refresh Rate?

The **refresh rate** of a screen is how many times per second the display updates the image.

- If your screen is **60 Hz**, it refreshes 60 times per second.
- If it's **120 Hz**, it refreshes 120 times per second (smoother animation).

---

## 🧠 What is Adaptive Refresh Rate (ARR)?

**Adaptive Refresh Rate (ARR)** means:

> The screen **automatically changes its refresh rate** depending on what is being shown — to **save battery** and reduce **jank** (stutter).

---

## 💡 Example:

Let's say your phone screen supports **120 Hz**.

| Situation | Normal Screen (Fixed 120 Hz) | ARR Screen (Adaptive) |
|---|---|---|
| Watching a video at 30 FPS | Screen still refreshes at 120 Hz | Screen refreshes at 30 Hz (saves power) |
| Reading a static article | Screen refreshes at 120 Hz | Screen slows to 10 or 24 Hz (saves power) |
| Playing a 60 FPS game | 120 Hz (wasted cycles) | Switches to 60 Hz (more efficient) |
| Scrolling fast | Uses 120 Hz | Uses 120 Hz (for smoothness) |

So with ARR:

- The **screen adapts** to match the content's **frame rate**.
- This saves **battery** and prevents **jumpy frames** (called "jank").

---

## 🛠️ How it works technically (in simple terms):

Imagine the display **blinks** (VSync) at **240 Hz** (every 4.16 ms), but it doesn't need to show a new picture every time.

If a video frame comes every 16.6 ms (60 Hz), the screen can:

- Wait 4 × 4.16 ms = 16.6 ms
- Then show the new frame

This is like **"only updating when needed"**, but using a **clock that ticks very fast** underneath.

So even though the screen can update fast, it **chooses to update less often** if the content doesn't need it.

## 📱 What's new in Android 15?

Android 15 adds **native support for ARR**. Here's what's added:

- New **APIs in the Hardware Composer (HWC)** layer.
- System tells the display when the **next frame is expected**.
- Helps the screen **plan** when to refresh and at what rate.

## 🧪 Testing Tools

- **TouchLatency app**: Let's you manually adjust frame rates and see how the display responds.
- **onRefreshRateChangedDebug()**: Tells you if the refresh rate changed, for debugging.

## 🔧 Terms Simplified:

| Term | Meaning (Simple) |
|---|---|
| `vsyncPeriod` | How often the screen *can* update |
| `minFrameIntervalNs` | Fastest refresh possible (smallest gap between frames) |
| `notifyExpectedPresent()` | System tells the screen "Next frame is coming at this time" |
| `frameIntervalNs` | The interval between each frame (e.g. 16.6ms for 60 Hz) |
| `VrrConfig` | Configuration for ARR support |
| `DisplayConfiguration.vrrConfig` | Enables ARR mode (if set), disables ARR if null |

## ✅ Benefits of ARR

| Benefit | How |
|---|---|
| 💡 Saves battery | Refreshes slower when not needed |
| 🧊 Smoother experience | No "mode switching", fewer frame drops |
| 🔋 More efficient | Keeps performance high only when necessary |

## ✅ Summary (One Line)

**ARR allows your screen to refresh only as fast as needed — smartly balancing performance and power.**

Let me know if you want a visual diagram or code example.

what is layer please explain in simple words and how it is related with surface

In simple words:

---

## ⬛ What is a Layer?

A **Layer** is like a **transparent sheet** where something is drawn (like a window, a button, a video frame, or part of your app UI).

Think of layers like pages in a flipbook — each page can have something drawn on it, and they are stacked one on top of another. What you **see on screen** is the result of **all visible layers** being combined (composited) together.

---

## 🖼️ What is a Surface?

A **Surface** is the **actual memory area (or buffer)** where pixels are drawn. It's like a **canvas** you can draw on using OpenGL, Vulkan, or Skia (Android's graphics engine).

---

## 🧩 So, how are they related?

A **Layer** is created from a **Surface + SurfaceControl**.

- The **Surface** holds the pixels (what you want to show).
- The **Layer** controls **how and where** that surface shows on screen — like position, size, transparency, order, etc.

  ✅ You draw to a Surface.
  ✅ Surface is attached to a Layer.
  ✅ SurfaceFlinger (Android's compositor) takes all layers and combines them into the final screen image.

---

## 🔁 Example:

Let's say your app has:

- A background image
- A video player window
- A chat bubble

Here's how it works:

| Content | Surface | Layer Behavior |
|---|---|---|
| Background img | Surface A | Bottom layer, full screen |
| Video Player | Surface B | On top of background |
| Chat Bubble | Surface C | Topmost, semi-transparent |

Each Surface (A, B, C) is drawn separately.
Each one has a Layer that tells SurfaceFlinger:

- Where to put it
- Whether it's transparent
- How big it is
- What Z-order (which one is on top)

Then **SurfaceFlinger** blends these layers to make the final image you see on screen.

---

### 📱 Real-world analogy:

Think of a phone screen like a sandwich:

- 🍞 Bottom bread: Background layer
- 🥩 Middle layers: Video, app UI
- 🧀 Top layer: Notifications or popups

Each ingredient is drawn separately but shown together as a whole — that's how **layers and surfaces** work in Android graphics.

---

Let me know if you'd like a diagram to go with it!

SurfaceControl, explain in simple words with example

Absolutely! Let's break down **`SurfaceControl`** in **very simple words** with a real-world analogy and example:

---

## 🎛️ What is `SurfaceControl`?

**`SurfaceControl`** is like a **remote control** for a layer on the screen.

- It doesn't draw anything itself.
- But it **controls how and where a Surface appears** on the screen.
- It controls things like:
  - position
  - size
  - visibility
  - layer stacking (Z-order)
  - transparency
  - rotation
  - cropping, etc.

---

## 🧱 How it fits in:

You usually have:

- A **Surface** → where your app draws pixels (like a canvas).
- A **SurfaceControl** → controls how that Surface is shown as a **Layer**.

Together, they form a **Layer**, and SurfaceFlinger puts all layers together to show the final screen.

---

## 📱 Real-world analogy:

Think of your phone as a **theater stage**.

- 🎨 **`Surface`** = the **actor's costume** (pixels/content).
- 🎚️ **`SurfaceControl`** = the **stage director's control panel** (controls spotlight, position, size).
- 🧱 **`Layer`** = the **actor on stage** (content + how it appears).

You (the app) give instructions like:

- "Put the video here"
- "Fade this in"
- "Move this up"
- "Hide that layer"

And SurfaceControl sends that info to **SurfaceFlinger**, which is like the backstage crew putting the full scene together.

---

# ✅ Example:

Let's say you want to show a video in the corner of the screen.

### Step-by-step:

1. Your app **creates a Surface** for the video output.
2. It creates a **SurfaceControl** attached to that Surface.
3. You use **`SurfaceControl.Transaction()`** to:
   - Set position: **`x = 100, y = 100`**
   - Set size: **`width = 640, height = 360`**
   - Set Z-order (on top of other things)

```java
SurfaceControl sc = new SurfaceControl.Builder()
    .setName("MyVideoLayer")
    .setBufferSize(640, 360)
    .build();

Surface surface = new Surface(sc);

new SurfaceControl.Transaction()
    .setLayer(sc, 10)       // Z-order
    .setPosition(sc, 100, 100)
    .show(sc)
    .apply();
```

Now your **video Surface is visible on screen** in the right place and size, thanks to SurfaceControl.

---

## 🧠 Key points to remember:

- **`SurfaceControl`** = controls the **Layer**.
- **`Surface`** = holds the actual pixels.
- Used for powerful control over how things are displayed (especially in system UI, media, games, multi-window apps, etc.)

---

Would you like a diagram showing this relationship?

How SurfaceControl is related to window manager?

Great question! Let's explain how `SurfaceControl` is related to the **Window Manager (WM)** in simple terms.

---

## 🧠 TL;DR (in very simple words):

- `WindowManager` is like the **boss** that manages **windows** from apps.
- `SurfaceControl` is like the **tool** used by WindowManager (and apps) to control how windows appear on screen.
- **WindowManager uses SurfaceControl** under the hood to show, move, resize, hide, or animate windows.

---

## 🎯 Think of it like this:

Imagine you're building a stage performance:

| Role | Real-world Equivalent |
|---|---|
| WindowManager | Stage Manager (organizes everything) |
| Window | Each actor's scene |
| SurfaceControl | Remote controls for lights, spotlight, position |
| Surface | The costumes and props on stage |
| SurfaceFlinger | The crew showing the full performance on stage (final screen) |

So the **WindowManager** tells **SurfaceControl**:

- "Show this app here"
- "Move this window to the top"
- "Hide this dialog"
- "Resize the split-screen view"

And then **SurfaceFlinger** puts it all together and renders it on the screen.

---

## 🔗 How exactly are they connected?

### Internally in Android:

- Every **Window** (like your app's activity, dialog, popup) has a **SurfaceControl** behind it.
- The **WindowManagerService (WMS)** and **WindowManagerPolicy** create and manage these windows.
- They use **SurfaceControl** to control:

    - layering (Z-order)
    - positioning
    - visibility
    - transformations (scale, rotate)
    - cropping
- These `SurfaceControl`s are all passed to **SurfaceFlinger**, which then composes them into the final screen image.

---

## 📱 Example Flow:

Say you launch the Calculator app:

1. WindowManager adds a new window for Calculator.
2. It creates a new `SurfaceControl` to represent that window's layer.
3. The Calculator app draws into its `Surface` (pixels).
4. WindowManager uses a `SurfaceControl.Transaction` to:
    - position the layer
    - set its visibility
    - apply animations
5. SurfaceFlinger takes this layer and draws it on screen.

---

## 🧩 Summary:

| Component | Role |
|---|---|
| `WindowManager` | Manages windows (placement, focus, input, etc.) |
| `SurfaceControl` | Controls **how each window appears** (position, size, visibility) |
| `Surface` | The actual pixel buffer the app draws into |
| `SurfaceFlinger` | The compositor that puts all layers together |

**So:**

- WindowManager creates and manages windows
- It uses SurfaceControl to **show and arrange them**
- Then SurfaceFlinger **renders everything**

---

Would you like a diagram of this flow?