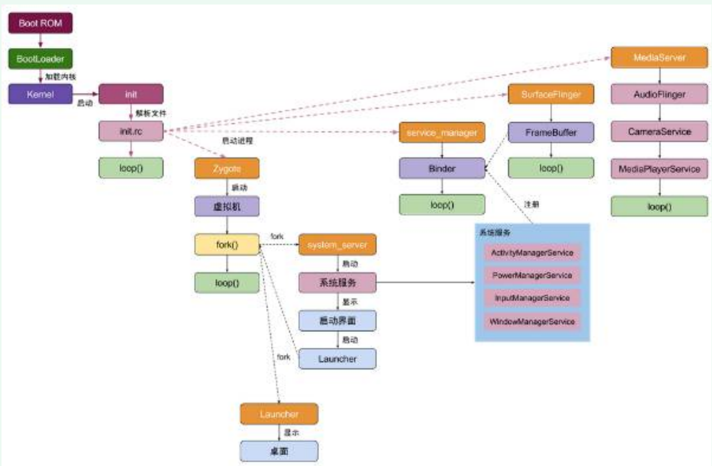
Surfaceflinger



Explain all the details in simple words

This diagram explains the **Android boot process** from powering on the device to launching the home screen (Launcher). Let's walk through it **step-by-step in simple terms**:

🧠 1. Power-On and Bootloader Stage

- **Boot ROM**
  ➤ This is built into the hardware. It starts automatically when you power on the phone.
- **BootLoader**
  ➤ Loads and starts the Android **Linux Kernel** into memory.
- **Kernel**
  ➤ The heart of the operating system. It sets up basic things like memory and hardware.

⚙️ 2. Init Process and Starting Services

- **Init**
  ➤ The first Android program started by the kernel.
  ➤ Reads the `init.rc` file to know what to do next.
- **init.rc**
  ➤ A script that tells `init` what services to start, like:
  - **Zygote**
  - **SurfaceFlinger**

- **MediaServer**
- **service_manager**

🧵 3. Core Processes Start

a) `service_manager`

➤ Manages and registers Android system services (like ActivityManager, PowerManager).

b) `SurfaceFlinger`

➤ Manages the screen display — how apps/layers are shown.

c) `MediaServer`

➤ Handles audio/video playback and camera services.

d) `Binder`

➤ Enables different Android components to talk to each other.

🧬 4. Zygote and Forking

- **Zygote**
  ➤ A special process that loads the Android framework and Java libraries.
  ➤ It is like a "template" used to quickly create other apps or system services.
- **fork()**
  ➤ A way to make a copy of Zygote when needed:
  - For **system_server** (which runs core system services)
  - For apps like **Launcher**

🎪 5. system_server and System Services

- **system_server**
  ➤ Starts essential **system services**, such as:
  - **ActivityManagerService** → Handles activities (UI screens).
  - **PowerManagerService** → Manages power and sleep.
  - **InputManagerService** → Handles touches and key presses.
  - **WindowManagerService** → Handles window positioning and focus.

🖼️ 6. Launcher (Home Screen)

- Zygote forks a new process to start the **Launcher app**.
- The Launcher is the first visible app you see (home screen).
- It allows the user to open apps and interact with the phone.

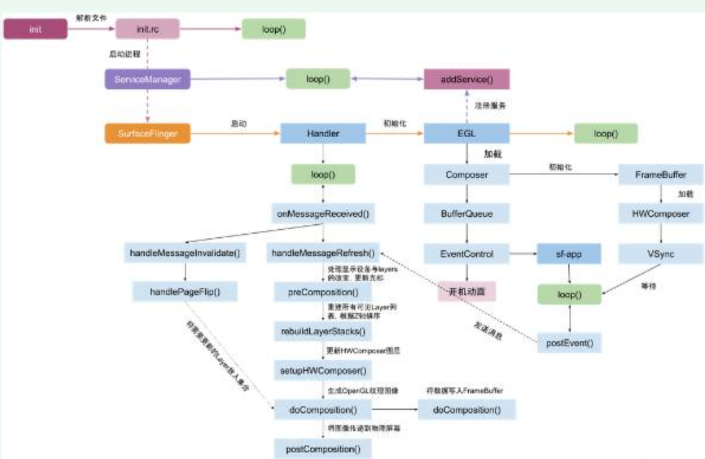
🔁 loop()

Wherever you see `loop()`:
➤ It means the process enters a **main loop**, waiting for messages/events to handle — this is how Android apps and services stay alive and responsive.

🧭 Summary (Flow)

1. Boot ROM → Bootloader → Kernel
2. Kernel → init → init.rc
3. init.rc → Starts: Zygote, service_manager, MediaServer, SurfaceFlinger
4. Zygote → fork system_server → start system services
5. Zygote → fork Launcher → show home screen

Let me know if you'd like this explained with real file names or logs (like `logcat`) from an actual Android device.



This second diagram focuses on **SurfaceFlinger**, the core Android system service responsible for **compositing and rendering the screen**. Let's break it down step-by-step in **simple words**, just like a story.

🔧 1. Boot Sequence (relevant parts)

From `init.rc`:

- `init.rc` tells the system to **start SurfaceFlinger**.
- It also starts **ServiceManager** and other services.
- These services run in a loop waiting for commands.

🎬 2. SurfaceFlinger Starts

SurfaceFlinger Initialization:

- **SurfaceFlinger** starts and:
  - **Registers itself with ServiceManager** using `addService()` so other services/apps can find it.
  - Sets up **EGL**, which is a library that helps SurfaceFlinger talk to **OpenGL ES or Vulkan** to render things.
  - Starts its own **loop()** to wait for screen update messages.

🧠 3. EGL & Rendering Setup

EGL Setup:

- Initializes rendering components:
  - **Composer**: Coordinates what should be drawn.
  - **BufferQueue**: Queue of images (layers) to be rendered.
  - **EventControl**: Manages animation timing (like boot animation).
  - **FrameBuffer**: Actual screen memory to write to.
  - **HWComposer**: Hardware abstraction for composition (blending images together).

🎡 4. Frame Update Flow — Handling Messages

SurfaceFlinger's Event Loop

- SurfaceFlinger runs a **Handler** loop that waits for events like:

- Something changed and needs to be redrawn (invalidate).
- It's time for a refresh (based on VSYNC signal).

📦 5. Message Handlers — How Drawing Happens

a) `handleMessageInvalidate()`

- Something changed, needs redraw.
- Calls `handlePageFlip()` to prepare next frame.

b) `handleMessageRefresh()`

- Called regularly (usually every 16ms for 60fps).
- Triggers a full frame redraw:
    1. **preComposition()** – Prepare layer list.
    2. **rebuildLayerStacks()** – Figure out which layers are visible and how.
    3. **setupHWComposer()** – Tell hardware composer how to blend them.
    4. **doComposition()** – Actually draw the layers.
        - If GPU is used → render OpenGL image.
        - If hardware composer is used → let it blend layers directly.
    5. **postComposition()** – Display the result on screen.

🔁 6. VSync, Animation, and Message Loop

VSync (Vertical Sync):

- Comes from display hardware.
- Tells SurfaceFlinger "time to draw a new frame".

sf-app (SurfaceFlinger animation app):

- A small part that handles animations (like boot animation).
- Uses `postEvent()` to send signals back into SurfaceFlinger's loop.

🧩 Summary – How the Screen Gets Updated

1. `init.rc` starts SurfaceFlinger.

2. SurfaceFlinger sets up rendering (EGL, BufferQueue, FrameBuffer).
3. VSync triggers frame update events.
4. SurfaceFlinger's **Handler** receives messages:
    - If something changed → `handleMessageInvalidate()`
    - On every VSync → `handleMessageRefresh()`
5. Refresh flow:
    - Compose layers → Draw using GPU or hardware → Output to screen.

📌 Real-World Analogy

Think of SurfaceFlinger like a **movie director**:

- It gets **frames (layers)** from apps.
- It **decides how to layer them**.
- Then it **asks the camera (GPU/HWC)** to shoot the final scene.
- Finally, it **displays** the shot on the screen.

Let me know if you'd like this diagram tied to real source files or log tags (like `SurfaceFlinger`, `BufferQueueProducer`, etc.).

```
... cc_binary {
   name: "surfaceflinger" ,
   defaults: [ "surfaceflinger_defaults" ],
   init_rc: [ "surfaceflinger.rc" ],
   srcs: [ "main_surfaceflinger.cpp" ],
   whole_static_libs: [
      "libsigchain" ,
   ],
   shared_libs: [
      "android.frameworks.displayservice@1.0" ,
       "android.hardware.configstore-utils" ,
       "android.hardware.configstore@1.0" ,
       "android.hardware.graphics.allocator@2.0" ,
       "libbinder" ,
       "libcutils" ,
       "libdisplayservicehidl" ,
       "libhidlbase" ,
       "libhidltransport" ,
```

```
        "liblayers_proto" ,
        "liblog" ,
        "libsurfaceflinger" ,
        "libtimestats_proto" ,
        "libutils" ,
    ],
    static_libs: [
        "libserviceutils" ,
        "libtrace_proto" ,
    ],
    ldflags: [ "-Wl,--export-dynamic" ],

    // TODO(b/71715793): These version-scripts are required due to the use of
    // whole_static_libs to pull in libsigchain. To work, the files had to be
    // locally duplicated from their original location
    // $ANDROID_ROOT/art/sigchainlib/
    multilib: {
        lib32: {
            version_script: "version-script32.txt" ,
        },
        lib64: {
            version_script: "version-script64.txt" ,
        },
    },
} ...
```

This `cc_binary` build rule describes how the **SurfaceFlinger binary** is built in Android using Soong (the Android.bp build system).

Let's explain each part **in simple terms**:

🏗️ `cc_binary { ... }`

Defines a **native (C/C++) binary** that will be compiled — in this case, it's for SurfaceFlinger.

🧱 Basic Properties

Field	Meaning
`name: "surfaceflinger"`	The name of the final binary. Installed as `/system/bin/surfaceflinger`.
`defaults: ["surfaceflinger_defaults"]`	Inherits common settings (like CFLAGS, include paths) from a default module.
`init_rc: ["surfaceflinger.rc"]`	Installs this init script so `init` can start SurfaceFlinger during boot.

Field	Meaning
`srcs: ["main_surfaceflinger.cpp"]`	This is the main source file that contains **`main()`** for SurfaceFlinger.

📦 Static Libraries

Field	Meaning
`whole_static_libs: ["libsigchain"]`	Links in **all** object files from this static library. **`libsigchain`** helps handle signal chaining (for e.g., signal handling in ART).
`static_libs: [...]`	Statically links in specific utility libraries (like **`libtrace_proto`** for tracing). These are linked once per usage.

🔗 Shared Libraries

These are **dynamically linked at runtime** and represent shared system services or utility libraries:

- **AIDL/HIDL Interfaces** (hardware or system services):
  - `android.frameworks.displayservice@1.0`
  - `android.hardware.configstore@1.0`
  - `android.hardware.graphics.allocator@2.0`
  - `libdisplayservicehidl`
- **System IPC & Utils**:
  - `libbinder` → Binder IPC
  - `libhidlbase`, `libhidltransport` → HIDL support
  - `libutils`, `libcutils` → Android base utilities
  - `liblog` → Logging
  - `libsurfaceflinger` → Main logic for SurfaceFlinger
  - `liblayers_proto`, `libtimestats_proto` → Protobuf definitions

These allow SurfaceFlinger to **interact with other system components**, allocate graphics buffers, and talk to hardware.

⚙️ Linker Flags

Field	Meaning
`ldflags: ["-Wl,--export-dynamic"]`	Tells the linker to **export all symbols** for dynamic linking or for tools like backtraces/profilers.

🔀 `multilib` & Version Scripts

Field	Meaning
`multilib`	Sets separate configuration for 32-bit (`lib32`) and 64-bit (`lib64`) builds.
`version_script`	These control which symbols are exported. Needed because `libsigchain` is statically linked with `whole_static_libs`, which can cause symbol conflicts.

> The comment mentions that these version scripts were copied locally from `art/sigchainlib/`.

📁 Final Output

This builds:

- A binary named `surfaceflinger`
- Includes HIDL, Binder, and rendering logic
- Uses an init script `surfaceflinger.rc` to start the process during boot
- Installed to the system partition and runs as part of Android's **graphics stack**

Let me know if you'd like to:

- Trace the startup from `main_surfaceflinger.cpp`
- See what's inside `surfaceflinger.rc`
- Understand how `libsurfaceflinger` is implemented

2. SurfaceFlinger Code Analysis [Android 11]

Code path: /frameworks/native/services/surfaceflinger/
The SurfaceFlinger binary is divided into the surfaceflinger executable file (main entry) and the libsurfaceflinger.so library file (function implementation), which is compiled from the main_surfaceflinger.cpp file. The Android.bp code module compilation configuration is as follows:

1. [Executable file - surfaceflinger]

You're absolutely right — in **Android 11**, the `SurfaceFlinger` binary is structured into two parts:

✅ 1. SurfaceFlinger Code Structure Overview

- 📁 **Path**: `/frameworks/native/services/surfaceflinger/`
- 🧩 **Components**:
  - **`main_surfaceflinger.cpp`** → This is the **entry point** (**`main()`** function).
  - **`libsurfaceflinger.so`** → Contains most of the **core logic**, such as rendering, composition, and interaction with HWC (Hardware Composer).

🏗️ 2. Compilation Overview – Android.bp

You already posted the **`cc_binary`** module for **`surfaceflinger`**. Let's summarize its role:

🧾 A. `main_surfaceflinger.cpp` (Executable Entry Point)

Type: `cc_binary`

What it does:

- Starts the SurfaceFlinger service by:
  - Setting up threads
  - Creating the **`SurfaceFlinger`** class object
  - Initializing EGL/GL and related graphics components
  - Registering the service via Binder
  - Entering the main event loop

Code snippet from `main_surfaceflinger.cpp`:

```cpp
int main(int, char**) {
    ...
    sp<SurfaceFlinger> flinger = new SurfaceFlinger();
    ...
    flinger->init();
    ...
    defaultServiceManager()->addService(...);
    ...
    IPCThreadState::self()->joinThreadPool();
}
```

📦 B. `libsurfaceflinger` (Shared Library)

Type: `cc_library_shared`

What it contains:

- The **actual SurfaceFlinger class** implementation.
- Handles:
  - Layer management
  - Composition strategy
  - HWComposer integration
  - BufferQueue handling
  - VSync
  - Transactions
  - Frame rendering pipeline

Key source files:

- `SurfaceFlinger.cpp` – the core class implementation
- `Layer.cpp` / `LayerBE.cpp` – represent compositable units (each app window or surface is a Layer)
- `RenderEngine/` – abstraction for OpenGL/Vulkan rendering
- `Scheduler/` – manages vsync and frame timing
- `DisplayHardware/` – abstracts physical display and vsync driver
- `CompositionEngine/` – coordinates layer composition, especially post Android 10/11 modularization

---

🔄 Relationship Between Executable and Library

Executable (`surfaceflinger`)	Shared Library (`libsurfaceflinger.so`)
Defines `main()`	Defines the `SurfaceFlinger` class
Calls `SurfaceFlinger::init()`	Implements rendering logic
Registers service	Implements service behavior

🧱 Modular Design Benefit

By splitting the entry (`main_surfaceflinger.cpp`) and implementation (`libsurfaceflinger`), Android:

- Allows other modules to reuse `libsurfaceflinger`
- Makes testing, maintenance, and modularization easier
- Keeps the `main()` logic minimal and focused

Would you like a full call flow diagram for `SurfaceFlinger::init()` and how it gets to composition (e.g., VSync → invalidate → refresh → draw)?

cc_library_shared,cc_binary
why we need both

Great question!

In Android's build system, using **both `cc_library_shared` and `cc_binary`** for something like SurfaceFlinger helps **separate concerns** between:

- the **executable entry point** (`cc_binary`)
- the **reusable logic and implementation** (`cc_library_shared`)

Let's break it down clearly:

✅ Why we need both `cc_binary` and `cc_library_shared`

Purpose	`cc_binary`	`cc_library_shared`
What it builds	A standalone executable (e.g., **`surfaceflinger`**)	A shared library (**`libsurfaceflinger.so`**)
Contains	Just **`main()`** and service registration logic	All the functional logic of SurfaceFlinger
Used by	Boot/init system (calls the binary)	Binary and possibly other modules or services
Reusable?	❌ No – it's an app or service entry point	✅ Yes – other processes can dynamically link to it

---

🧱 Real Example: SurfaceFlinger

`cc_binary` → `surfaceflinger`

```bp
cc_binary {
    name: "surfaceflinger",
    srcs: ["main_surfaceflinger.cpp"],
    shared_libs: ["libsurfaceflinger", ...]
}
```

- Contains the **`main()`** function.
- Instantiates **`SurfaceFlinger`** class.
- Starts Binder service and joins the main loop.

`cc_library_shared` → `libsurfaceflinger.so`

```
bp

cc_library_shared {
    name: "libsurfaceflinger",
    srcs: ["SurfaceFlinger.cpp", "Layer.cpp", ...]
}
```

- Implements all the functionality.
- Provides **`SurfaceFlinger::init()`**, **`SurfaceFlinger::handleMessageRefresh()`**, etc.
- Can be reused/tested independently.

🔁 Analogy

Think of **`cc_library_shared`** as a **car engine**, and **`cc_binary`** as the **car**:

- You build the **engine separately** so it can be reused in other cars (or tested by itself).
- The **car binary** (**`main()`**) knows how to **start the engine**, register with traffic (Binder), and drive (event loop).

📦 Benefits of Splitting

Benefit	Description
🔁 Reusability	Other services (like **`SurfaceFlinger_test`**, or tools like **`dumpstate`**) can link against **`libsurfaceflinger`**.
🧪 Testability	You can unit test **`libsurfaceflinger`** functions without needing to launch the whole binary.
🔧 Modularity	Keeps the binary small and clean. Complex code stays in the shared library.
🔄 Maintenance	If logic changes, you often only update the **`.so`**, not the full binary.

TL;DR

- **`cc_binary`**: **Runs the process** (has **`main()`**).
- **`cc_library_shared`**: **Implements the logic** (reused by binary and others).
- We use **both** so logic is cleanly separated from process entry, reused easily, and maintained modularly.

Let me know if you want a breakdown of the call flow starting from **`main()`** → **`SurfaceFlinger::init()`** → frame drawing!

For what purpose will others use libsurfaceflinger.so

Excellent question — and a key one for understanding Android's graphics architecture.

🧩 Why Others Might Use `libsurfaceflinger.so`

While **`surfaceflinger` binary** is the actual service that runs and handles display composition, the **`libsurfaceflinger.so` shared library** exposes **core logic and utilities** that other parts of the system may **link to**, including:

✅ 1. Testing and Debugging Tools

- **SurfaceFlinger test binaries** (like `SurfaceFlinger_test`):
  - Link to `libsurfaceflinger.so` to test internal functions.
  - Useful for unit testing composition logic without launching the full system.

✅ 2. System UI or Shell Components

- Android's **Shell**, **SystemUI**, or **WindowManager** may:
  - Use `libsurfaceflinger` symbols to query or manipulate surface or display state.
  - This is rare and tightly controlled (mostly indirect via Binder IPC), but may still link certain internal headers for debugging or analysis.

✅ 3. Debugging/Profiling Tools

- Tools like `dumpsys`, `SurfaceFlinger::dump()`:
  - The binary uses public `dump()` APIs implemented in `libsurfaceflinger.so` to collect surface, layer, and composition info.
  - These symbols are reused for command-line and UI-level debugging.

Example:

```bash
adb shell dumpsys SurfaceFlinger
```

✅ 4. Binder IPC Glue Code

Even though most components talk to **`SurfaceFlinger` via Binder IPC**, the IPC stub/proxy code may depend on types or utility functions provided by `libsurfaceflinger`.

- For example:
  - `ISurfaceComposer` or `DisplayDevice` types may be declared in shared headers linked with `libsurfaceflinger`.

✅ 5. Layer Management Utilities

If some system service or tool (e.g. `dumpsys`, `simpleperf`, `SurfaceTracing`) wants to:

- Inspect layer trees
- Analyze frame rendering
- Log time spent per layer
  Then it may reuse `libsurfaceflinger.so` to avoid duplicating logic.

✅ 6. Vendor Extensions / Debug Builds

In debug or custom vendor builds:

- OEMs (like Samsung, Xiaomi) may extend or patch SurfaceFlinger.
- Instead of modifying the binary, they may dynamically hook or wrap around the `.so` to:
  - Inject frame capture
  - Customize composition logic
  - Export additional logs

🧪 Summary of Consumers of `libsurfaceflinger.so`

Module	Why it uses `libsurfaceflinger.so`
`surfaceflinger` (binary)	Main service
`dumpsys`	Access layer and frame state
`SurfaceFlinger_test`	Unit testing
`simpleperf`, **`SurfaceTracing`**	Performance tracing
Vendor tools / OEM patches	Custom diagnostics or frame capture
Debug UI / Shell tools	Read internal Surface state

⚠️ Note:

Most apps or services **do not directly link `libsurfaceflinger.so`** — they use **Binder IPC** via **`ISurfaceComposer`** and **`ISurfaceComposerClient`**. But **internal tools and tests** do link the library to reuse shared logic.

Would you like to see a list of functions exposed by **`libsurfaceflinger.so`** (e.g., from symbol dumps) or how **`ISurfaceComposer`** is used across modules?

```
int main( int , char ** ) {
    signal(SIGPIPE, SIG_IGN);

    // Starting from 8.0, Android provides the HIDL mechanism, which standardizes the original direct JNI->Native->HAL interface call method into a HIDL service/client interaction method.
     // This method standardizes and unifies the calling mechanism of the Android Framework and HAL. However, from a project perspective, this calling method has a performance overhead and will take more time than the direct calling method.
    hardware::configureRpcThreadpool(1 /* maxThreads */ ,
            false  /* callerWillJoin */ );

    startGraphicsAllocatorService();

    // When SF is launched in its own process, limit the number of
     // binder threads to 4.
    ProcessState::self()->setThreadPoolMaxThreadCount(4 );

    // start the thread pool
    sp<ProcessState> ps(ProcessState::self());
    ps- > startThreadPool();

    // Create a SurfaceFlinger object, pointed to by a strong pointer.
     // SurfaceFlinger inherits the RefBase class, so once the new object is assigned to the sp pointer, the onFirstRef method of the SurfaceFlinger class will be called immediately.
     // instantiate surfaceflinger
    sp<SurfaceFlinger> flinger = surfaceflinger::createSurfaceFlinger();

    setpriority(PRIO_PROCESS, 0 , PRIORITY_URGENT_DISPLAY);

    set_sched_policy( 0 , SP_FOREGROUND);

    // Put most SurfaceFlinger threads in the system-background cpuset
     // Keeps us from unnecessarily using big cores
     // Do this after the binder thread pool init
```

```
    if (cpusets_enabled()) set_cpuset_policy(0 , SP_SYSTEM);

    // SurfaceFlinger class formally initialized
     // initialize before clients can connect
    flinger-> init();

    // SurfaceFlinger registers the Binder service with ServiceManager,
     // so that other processes can get the SurfaceFlinger service through
getService+SERVICE_NAME, and then can communicate with the SurfaceFlinger class through
Binder.
     // publish surface flinger
    sp<IServiceManager> sm(defaultServiceManager());
    sm ->addService(String16(SurfaceFlinger::getServiceName()), flinger, false ,
                   IServiceManager::DUMP_FLAG_PRIORITY_CRITICAL |
IServiceManager::DUMP_FLAG_PROTO);

    // The new DisplayService() method inside calls the HIDL definition interface
Return<sp<IDisplayEventReceiver >> getEventReceiver() override;
    startDisplayService(); // dependency on SF getting registered above

    if (SurfaceFlinger::setSchedFifo( true ) != NO_ERROR) {
        ALOGW( "Couldn't set to SCHED_FIFO: %s" , strerror(errno));
    }

    // SurfaceFlinger class enters the main loop (note that the SurfaceFlinger class does not
inherit the Threads class and does not follow the interface execution order of the Threads class)
     // run surface flinger in this thread
    flinger-> run();

    return 0 ;
}
```

This is the **`main()` function** of the **`surfaceflinger`** binary from **Android 11**, defined in **`main_surfaceflinger.cpp`**. It's the **entry point** for the SurfaceFlinger system service. Let's break it down line by line in **simple language** with comments on what each part does.

🧠 Step-by-step Explanation

```cpp
int main(int, char**) {
```

> Entry point. Parameters unused here.

```cpp
signal(SIGPIPE, SIG_IGN);
```

Ignore **SIGPIPE**, which can occur if a write is done to a closed socket or pipe. Prevents the process from being killed.

```cpp
hardware::configureRpcThreadpool(1 /* maxThreads */, false /* callerWillJoin */);
```

This sets up the **HIDL Binder threadpool** for handling remote HAL calls.

- `1`: Max 1 thread handles incoming HIDL calls.
- `false`: Main thread doesn't join the HIDL pool (it's separate).

```cpp
startGraphicsAllocatorService();
```

Starts the **graphics memory allocator HAL service** (Gralloc), which handles buffer allocations shared between SurfaceFlinger and GPU/HWC.

```cpp
ProcessState::self()->setThreadPoolMaxThreadCount(4);
```

Limits the **Binder IPC threadpool** to 4 threads. This is important because SF is sensitive to scheduling overhead and latency.

```cpp
sp<ProcessState> ps(ProcessState::self());
ps->startThreadPool();
```

Starts the **Binder IPC thread pool** so SurfaceFlinger can receive IPC calls from clients like WindowManager.

```cpp
sp<SurfaceFlinger> flinger = surfaceflinger::createSurfaceFlinger();
```

Creates the main **SurfaceFlinger object** using a smart pointer `sp<>`.

- Because SurfaceFlinger inherits `RefBase`, its `onFirstRef()` method will be called right after creation.

```cpp
setpriority(PRIO_PROCESS, 0, PRIORITY_URGENT_DISPLAY);
```

Boosts the priority of this process so it gets **high priority scheduling** — important for rendering to happen on time (avoid jank/stutter).

```cpp
set_sched_policy(0, SP_FOREGROUND);
```

Moves this process to the **foreground scheduling group**, giving it more CPU time compared to background processes.

```cpp
if (cpusets_enabled()) set_cpuset_policy(0, SP_SYSTEM);
```

If **CPUsets** (CPU partitioning feature) are enabled:

- Place SF into the **system-background cpuset**.
- Helps avoid using **big CPU cores unnecessarily** (power saving).

```cpp
flinger->init();
```

Calls the `init()` method of the `SurfaceFlinger` class:

- Initializes EGL, DisplayDevices, HWComposer, Scheduler, etc.
- Must be done **before clients connect** (e.g., SurfaceComposerClient, DisplayService).

```cpp
sp<IServiceManager> sm(defaultServiceManager());
sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, false,
               IServiceManager::DUMP_FLAG_PRIORITY_CRITICAL | IServiceManager::DUMP_FLAG_PROTO);
```

Registers **SurfaceFlinger** as a **Binder service** named **`"SurfaceFlinger"`**:

- Other processes (like **`WindowManager`**, **`SystemUI`**) get access using **`getService("SurfaceFlinger")`**.
- They can send commands like **`createSurface()`**, **`setLayerStack()`**, etc.

```cpp
startDisplayService(); // dependency on SF getting registered above
```

Starts the **`DisplayService`**, which uses SF to handle display events.

- Relies on SF already being registered to function properly.

```cpp
if (SurfaceFlinger::setSchedFifo(true) != NO_ERROR) {
    ALOGW("Couldn't set to SCHED_FIFO: %s", strerror(errno));
}
```

Tries to enable **real-time FIFO scheduling** for SurfaceFlinger.

- Helps ensure strict timing and deadline for frame composition.
- Warns if it fails, but continues running.

```cpp
flinger->run();
```

Enters **SurfaceFlinger's main loop**:

- Waits for **VSYNC, buffer updates, composition requests**.
- Handles messages using a **Handler-based event loop**.
- Keeps rendering frames until the system shuts down.

```cpp
return 0;
```

Exit code (won't be reached under normal operation).

🔄 Overall Flow Summary

Phase	What Happens
🔧 Setup	Signal handling, thread pools, HIDL RPC threadpool
⚙️ Initialization	Create SF object, initialize graphics stack
📡 Service Registration	Register SF to ServiceManager
🔗 Start Dependent Services	Start DisplayService
🧠 Real-time Scheduling	Try to boost scheduler priority
🔁 Event Loop	Enter **`run()`** and handle screen composition, messages, vsync

---

Would you like a visual diagram showing how this **`main()`** connects to **`SurfaceFlinger::init()`**, the VSync loop, and the display pipeline?

hardware::configureRpcThreadpool(1 /* maxThreads */, false /* callerWillJoin */);
and ProcessState::self()->setThreadPoolMaxThreadCount(4);
why these two

Excellent question! These two lines both set up **threadpools**, but for **different Binder IPC systems** in Android.

Let me explain them clearly and simply:

🧵 1. `hardware::configureRpcThreadpool(...)`

```cpp
hardware::configureRpcThreadpool(1, false);
```

📌 Purpose:

This sets up the **HIDL Binder RPC threadpool** (used for hardware services).

📦 Used for:

- HIDL HAL services (e.g., Gralloc, Composer HAL)
- SurfaceFlinger may serve as a **HIDL client or server**

🧠 Meaning of arguments:

- **`1`**: Use **1 thread** to handle incoming HIDL method calls.
- **`false`**: Main thread **will not join** this threadpool.

Think of it as: “How many threads can HIDL use to handle calls from other processes?”

🧵 2. `ProcessState::self()->setThreadPoolMaxThreadCount(4);`

```cpp
ProcessState::self()->setThreadPoolMaxThreadCount(4);
```

📌 Purpose:

This sets up the **AIDL Binder threadpool** — used for **Framework <-> System** communication.

📦 Used for:

- Classic Android Binder calls (e.g., **`SurfaceComposerClient -> SurfaceFlinger`**)
- SurfaceFlinger is a **Binder service** used by **`WindowManager`**, **`SystemUI`**, etc.

🧠 What it does:

- Allows **up to 4 threads** to handle incoming **AIDL Binder IPCs**.

  This means SurfaceFlinger can serve **up to 4 client requests concurrently** via Binder.

🔄 Summary Table

Function	Purpose	For Which IPC?	Who Uses It
`hardware::configureRpcThreadpool(1, false)`	Sets up HIDL thread pool	**HIDL Binder**	HAL calls (like Gralloc, Display HAL)
`setThreadPoolMaxThreadCount(4)`	Sets up AIDL thread pool	**AIDL Binder**	Framework clients (like WindowManager, SystemUI)

💡 Why both?

SurfaceFlinger talks to:

- **HALs via HIDL** → Needs HIDL threadpool (**`configureRpcThreadpool`**)
- **Apps/System via AIDL Binder** → Needs Binder threadpool (**`setThreadPoolMaxThreadCount`**)

Without both, it wouldn't be able to:

- Allocate buffers or interact with HWComposer HAL (HIDL)
- Serve system-level clients like `WindowManager` or `SurfaceControl` (AIDL)

Let me know if you'd like a diagram showing how SurfaceFlinger connects to other components via HIDL vs AIDL!

multilib: {
    lib32: {
        version_script: "version-script32.txt" ,
    },
    lib64: {
        version_script: "version-script64.txt" ,
    },
  },
why this?

Great question! Let's break this down simply.

🔧 `multilib` in `Android.bp` files

```bp
multilib: {
    lib32: {
        version_script: "version-script32.txt",
    },
    lib64: {
        version_script: "version-script64.txt",
    },
},
```

✅ Purpose:

To **build both 32-bit and 64-bit versions** of a native library, **each with their own version script**.

💡 Why do we need different version scripts?

📦 `version_script.txt`

A version script controls **which symbols** are exported from a `.so` file.

This is useful to:

- Hide internal/private functions
- Reduce binary size
- Prevent symbol conflicts
- Maintain ABI stability

🧍 Why separate for 32-bit and 64-bit?

Because:

- Symbol layouts may differ between architectures
- You might want to **expose or hide different symbols** on 32-bit vs 64-bit
- For compatibility or performance reasons, the ABI might differ slightly

---

🔄 What happens with this config?

The build system will:

1. Compile `libsomething.so` for 32-bit using `version-script32.txt`
2. Compile `libsomething.so` for 64-bit using `version-script64.txt`
3. Each version will only expose symbols defined in its respective script

---

📌 When is this used?

- System libraries that must support both 32-bit and 64-bit apps
- Libraries that are loaded by both 32-bit and 64-bit processes (e.g. `libsurfaceflinger.so`)
- Ensuring ABI compatibility across both architectures

🧠 Summary:

Field	Meaning
`multilib`	Indicates per-architecture config
`lib32.version_script`	Symbol export list for 32-bit `.so`
`lib64.version_script`	Symbol export list for 64-bit `.so`

This allows fine-grained control over what each version of the library exports.

Let me know if you want to see what's typically inside a version script!

2. [Dynamic library - libsurfaceflinger.so]
The compilation configuration of the Android.bp code module is as follows:

```
... cc_library_shared {
    name: "libsurfaceflinger" ,
    defaults: [ "libsurfaceflinger_defaults" ],
    cflags: [
        "-fvisibility=hidden" ,
         "-Werror=format" ,
         "-DREDUCE_VIDEO_WORKLOAD" ,
         "-DUSE_AML_HW_ACTIVE_MODE" ,
    ],
    srcs: [
        ":libsurfaceflinger_sources" ,
    ],
    logtags: [ "EventLog/EventLogTags.logtags" ],
    include_dirs: [
        "frameworks/native/vulkan/vkjson" ,
         "frameworks/native/vulkan/include" ,
         "hardware/amlogic/gralloc/amlogic" ,
         "hardware/amlogic/hwcomposer/tvp" ,
         "hardware/amlogic/gralloc" ,
    ],
    static_libs: [
        "libomxutils_static@2" ,
         "libamgralloc_ext_static@2" ,
    ],
    cppflags: [
        "-fwhole-program-vtables", // requires ThinLTO
    ],
    lto: {
        thin: true ,
    },
} ...
```

Let’s break this `cc_library_shared` block for `libsurfaceflinger.so` into **simple parts**, and explain **why** each is used.

📦 Overview:

This is the build rule in `Android.bp` to compile the shared library `libsurfaceflinger.so`.

This is the core library that implements SurfaceFlinger — Android's compositor service — and is used both internally (by the `surfaceflinger` binary) and externally (by other system components).

🔍 Breakdown of fields:

✅ `name: "libsurfaceflinger"`

Defines the name of the `.so` (shared object) library.

✅ `defaults: [ "libsurfaceflinger_defaults" ]`

Includes common settings from a default group (likely includes shared `srcs`, flags, etc.)

✅ `cflags: [ ... ]`

Compiler flags for **C/C++ compilation**:

```c
-fvisibility=hidden
```

→ Hide all symbols by default (only export those explicitly marked). Helps with:

- Smaller binary
- Preventing symbol conflicts

```c
-Werror=format
```

→ Makes format string warnings into **errors** (e.g., `printf("%d", str)` will fail to compile).

```c
-DREDUCE_VIDEO_WORKLOAD
-DUSE_AML_HW_ACTIVE_MODE
```

→ These are **macros** (defines) used in the code to enable Amlogic-specific optimizations, like:

- Reducing load for video layers
- Using hardware-specific power-saving or performance modes

✅ `srcs: [ ":libsurfaceflinger_sources" ]`

Includes source files from a **named source group**. Instead of listing `.cpp` files here, it reuses another group (likely declared elsewhere in the file).

✅ `logtags: [ "EventLog/EventLogTags.logtags" ]`

Used for Android's `EventLog`. This file defines tags for structured logging in logs.

✅ `include_dirs: [ ... ]`

Adds extra **header file search paths** — these are needed for external dependencies:

- Vulkan support: `"frameworks/native/vulkan/..."`
- Amlogic hardware-specific code: `"hardware/amlogic/..."`

This shows SurfaceFlinger is integrated with Vulkan and customized for Amlogic chips.

✅ `static_libs: [ ... ]`

These are **static libraries** that will be **linked into** `libsurfaceflinger.so` at build time:

```text
libomxutils_static@2       → OMX utilities for media
libamgralloc_ext_static@2  → Amlogic Gralloc extension
```

These are architecture-specific helpers for memory and buffer handling.

✅ `cppflags: [ "-fwhole-program-vtables" ]`

Enables whole-program vtable optimization (makes C++ virtual function dispatch more efficient). Requires ThinLTO.

✅ `lto: { thin: true }`

Enables **Thin Link-Time Optimization (LTO)**:

- Improves performance by optimizing across files
- ThinLTO is **faster and uses less memory** than full LTO

✅ Summary

Field	What It Does
`name`	Name of the shared library (`libsurfaceflinger.so`)
`defaults`	Imports common settings
`cflags`	Compiler flags for warnings, symbol visibility, and macros
`srcs`	Source files (via a group)
`logtags`	Tags for event logging
`include_dirs`	Extra header paths (e.g., Vulkan, Amlogic)
`static_libs`	Static `.a` libraries to link
`cppflags`	Extra C++ optimization flags
`lto.thin`	Enables Thin Link-Time Optimization

This build config ensures:

- A secure and optimized binary
- Hardware-specific support
- Compatibility with other system components

Let me know if you want me to explain the `.bp` for the SurfaceFlinger executable (`cc_binary`) too!

srcs: [":libsurfaceflinger_sources"]
logtags: ["EventLog/EventLogTags.logtags"]
lto: { thin: true }

explain these in simple words

Here's a simple explanation for each of these Android.bp fields in the `cc_library_shared` module for `libsurfaceflinger`:

✅ `srcs: [ ":libsurfaceflinger_sources" ]`

- **What it means**: This includes source files (like `.cpp`) for compilation.
- **Why the colon** `:`: It's referencing another **module** named `libsurfaceflinger_sources`, not just raw files.
- **That module** contains the actual source file list (`srcs`) used to build `libsurfaceflinger`.

Simple analogy: It's like saying:

"Get the list of source files defined in the `libsurfaceflinger_sources` recipe and use them here."

✅ `logtags: [ "EventLog/EventLogTags.logtags" ]`

- **What it means**: It adds Android **logtag definitions**, which are used to generate `EventLog` constants.
- These tags are used when logging system events (e.g., via `ALOG_EVENT()`).
- Android will auto-generate a header file (e.g., `EventLogTags.h`) from this `.logtags` file.

Why: To map log event IDs to names consistently, like:

```cpp
LOG_EVENT_INT(MY_EVENT_ID, 123);
```

✅ `lto: { thin: true }`

- **LTO**: Link-Time Optimization — it makes your final `.so` smaller and faster by optimizing across all source files during final linking.
- **thin: true**: Use **ThinLTO**, a faster and more incremental version of LTO.

Simple explanation: Enables smarter compiler optimization *after* building the object files — but `thin` keeps build times reasonable.

Let me know if you want to see how `libsurfaceflinger_sources` is defined or what `logtags` look like.

The createSurfaceFlinger() mentioned above will call new SurfaceFlinger(), and then execute to: onFirstRef():

void SurfaceFlinger::onFirstRef()
{
  mEventQueue.init(this);
}
    In onFirstRef(), the Handler is created and initialized:
/frameworks/native/services/surfaceflinger/Scheduler/MessageQueue.cpp

Copy code
// MessageQueue.cpp
void MessageQueue::init(const sp<SurfaceFlinger>& flinger)

```
{
   mFlinger = flinger;
   mLooper = new Looper( true );
   mHandler = new Handler(* this );
}
```
Copy code

Then it will execute to SurfaceFlinger::init(), the main function of this method is:

Initializing EGL
Creating HWComposer
Initialize non-virtual display
Start the EventThread thread
Start boot animation
Copy code

```
// Do not call property_set on main thread which will be blocked by init
 // Use StartPropertySetThread instead.
void SurfaceFlinger::init() {
   ALOGI(   "SurfaceFlinger's main thread ready to run. "
           "Initializing graphics H/W..." );
   Mutex::Autolock _l(mStateLock);

  // Set the CompositionEngine property and create a RenderEngine object
    // Get a RenderEngine for the given display / config (can't fail)
    // TODO(b/77156734): We need to stop casting and use HAL types when possible.
    // Sending maxFrameBufferAcquiredBuffers as the cache size is tightly tuned to single-
display.
   mCompositionEngine-> setRenderEngine(renderengine::RenderEngine::create(
           renderengine::RenderEngineCreationArgs::Builder()
               .setPixelFormat(static_cast <int32_t> (defaultCompositionPixelFormat))
               .setImageCacheSize(maxFrameBufferAcquiredBuffers)
               .setUseColorManagerment(useColorManagement)
               .setEnableProtectedContext(enable_protected_contents( false ))
               .setPrecacheToneMapperShaderOnly( false )
               .setSupportsBackgroundBlur(mSupportsBlur)
               .setContextPriority(useContextPriority
                       ? renderengine::RenderEngine::ContextPriority::HIGH
                       : renderengine::RenderEngine::ContextPriority::MEDIUM)
               .build()));
   mCompositionEngine -> setTimeStats(mTimeStats);

   LOG_ALWAYS_FATAL_IF(mVrFlingerRequestsDisplay,
           "Starting with vr flinger active is not currently supported." );

       //Create a HWComposer object and pass in a name attribute, then set the object properties
through mCompositionEngine->setHwComposer.
   mCompositionEngine- >
```

```
setHwComposer(getFactory().createHWComposer(getBE().mHwcServiceName));
    mCompositionEngine- >getHwComposer().setConfiguration( this ,
getBE().mComposerSequenceId);

        //processDisplayHotplugEventsLocked(); //Process any initial hotplug and resulting display
changes
        . //This method mainly calls initScheduler(displayId);
     // Process any initial hotplug and resulting display changes.
    processDisplayHotplugEventsLocked();
     const auto display = getDefaultDisplayDeviceLocked();
    LOG_ALWAYS_FATAL_IF( !display, "Missing internal display after registering composer
callback." );
    LOG_ALWAYS_FATAL_IF( !getHwComposer().isConnected(*display-> getId()),
                        "Internal display is disconnected." );

    if (useVrFlinger) {
        auto vrFlingerRequestDisplayCallback = [ this ](bool requestDisplay) {
             // This callback is called from the vr flinger dispatch thread. We
             // need to call signalTransaction(), which requires holding
             // mStateLock when we're not on the main thread. Acquiring
             // mStateLock from the vr flinger dispatch thread trigger might a
             // deadlock in surface flinger (see b/66916578), so post a message
             // to be handled on the main thread instead.
            static_cast< void >(schedule([= ] {
                ALOGI( "VR request display mode: requestDisplay=%d" , requestDisplay);
                mVrFlingerRequestsDisplay = requestDisplay;
                signalTransaction();
            }));
        };
        mVrFlinger = dvr::VrFlinger::Create(getHwComposer().getComposer(),
                                            getHwComposer()
                                                    .fromPhysicalDisplayId( *display-> getId())
                                                    .value_or( 0 ),
                                            vrFlingerRequestDisplayCallback);
        if (! mVrFlinger) {
            ALOGE( "Failed to start vrflinger" );
        }
    }

    // initialize our drawing state
    mDrawingState = mCurrentState;

    // set initial conditions (eg unblank default device)
    initializeDisplays();

    char primeShaderCache[PROPERTY_VALUE_MAX];
```

```
    property_get( "service.sf.prime_shader_cache", primeShaderCache, "1" );
     if (atoi(primeShaderCache)) {
        getRenderEngine().primeCache();
    }

    // Inform native graphics APIs whether the present timestamp is supported:

    const bool presentFenceReliable =
            ! getHwComposer().hasCapability(hal::Capability::PRESENT_FENCE_IS_NOT_RELIABLE);
    mStartPropertySetThread = getFactory().createStartPropertySetThread(presentFenceReliable);

    if (mStartPropertySetThread->Start() != NO_ERROR) {
        ALOGE( "Run StartPropertySetThread failed!" );
    }

    ALOGV( "Done initializing" );
}
```

Copy code

First, let's see how to create HWComposer:frameworks/native/services/surfaceflinger/SurfaceFlingerDefaultFactory.cpp

```
// make_unique is equivalent to new (can replace new and does not require a delete pointer, which helps with code management).
std::unique_ptr<HWComposer> DefaultFactory::createHWComposer( const std::string& serviceName) {
     return std::make_unique<android::impl::HWComposer> (serviceName);
}
```

Initialize CompositionEngine:

```
std::unique_ptr<compositionengine::CompositionEngine>
DefaultFactory::createCompositionEngine() {
      return compositionengine::impl::createCompositionEngine();
}
```

The CompositionEngine object is also created through make_unique:

```
std::unique_ptr<compositionengine::CompositionEngine> createCompositionEngine() {
      return std::make_unique<CompositionEngine> ();
}
```

Go back to flinger->init() processDisplayHotplugEventsLocked(); to handle the results of any initial hot plug and display changes. In this method, the main call is initScheduler(displayId):

Copy code

```
void SurfaceFlinger::initScheduler(DisplayId primaryDisplayId) {
     if (mScheduler) {
         // In practice it's not allowed to hotplug in/out the primary display once it's been
```

```
        // connected during startup, but some tests do it, so just warn and return.
       ALOGW("Can't re-init scheduler" );
        return ;
    }

    auto currentConfig =
HwcConfigIndexType(getHwComposer().getActiveConfigIndex(primaryDisplayId));
    mRefreshRateConfigs =
            std::make_unique <scheduler::RefreshRateConfigs> (getHwComposer().getConfigs(
                                                        primaryDisplayId),
                                                currentConfig);
    mRefreshRateStats =
            std::make_unique <scheduler::RefreshRateStats>(*mRefreshRateConfigs, * mTimeStats,
                                              currentConfig, hal::PowerMode::OFF);
    mRefreshRateStats -> setConfigMode(currentConfig);

    mPhaseConfiguration = getFactory().createPhaseConfiguration(* mRefreshRateConfigs);

        // Create a Scheduler object
     // start the EventThread
    mScheduler =
            getFactory().createScheduler([ this ](bool enabled) { setPrimaryVsyncEnabled(enabled); },
                                 *mRefreshRateConfigs, * this );
     //Create app link and sf link
     mAppConnectionHandle =
            mScheduler ->createConnection("app", mPhaseConfiguration->
getCurrentOffsets().late.app,
                                impl::EventThread::InterceptVSyncsCallback());
    mSfConnectionHandle =
            mScheduler ->createConnection("sf", mPhaseConfiguration-> getCurrentOffsets().late.sf,
                                [ this ](nsecs_t timestamp) {
                                    mInterceptor -> saveVSyncEvent(timestamp);
                                });

    mEventQueue ->setEventConnection(mScheduler->
getEventConnection(mSfConnectionHandle));
    mVSyncModulator.emplace( * mScheduler, mAppConnectionHandle, mSfConnectionHandle,
                        mPhaseConfiguration -> getCurrentOffsets());

    mRegionSamplingThread =
             new RegionSamplingThread(* this , * mScheduler,
                            RegionSamplingThread::EnvironmentTimingTunables());
    // Dispatch a config change request for the primary display on scheduler
     // initialization, so that the EventThreads always contain a reference to a
     // prior configuration.
     //
```

```
    // This is a bit hacky, but this avoids a back-pointer into the main SF
     // classes from EventThread, and there should be no run-time binder cost
     // anyway since there are no connected apps at this point.
    const nsecs_t vsyncPeriod =
            mRefreshRateConfigs -> getRefreshRateFromConfigId(currentConfig).getVsyncPeriod();
    mScheduler -> onPrimaryDisplayConfigChanged(mAppConnectionHandle,
primaryDisplayId.value,
                                          currentConfig, vsyncPeriod);
}
Copy code
       See the links of app and sf in detail:

Copy code
Scheduler::ConnectionHandle Scheduler::createConnection(
         const  char * connectionName, nsecs_t phaseOffsetNs,
        impl::EventThread::InterceptVSyncsCallback interceptCallback) {
     auto vsyncSource = makePrimaryDispSyncSource(connectionName, phaseOffsetNs);
     auto eventThread = std::make_unique<impl::EventThread> (std::move(vsyncSource),
std::move(interceptCallback));
      return createConnection(std::move(eventThread));
}
Copy code
       You can see that the DispSyncSource object is created, and the construction method passes
in four values, dispSync object, phaseOffset offset, traceVsync is true, and name is app or sf

Copy code
  DispSyncSource::DispSyncSource(DispSync* dispSync, nsecs_t phaseOffset, bool traceVsync,
                                 const  char * name)
        : mName(name),
          mValue(base::StringPrintf( "VSYNC-%s", name), 0), // mValue is assigned a value. The
VSYNC-app VSYNC-sf label we see on systrace is it
          mTraceVsync(traceVsync),   // mTraceVsync is true, in the onDispSyncEvent method
          mVsyncOnLabel(base::StringPrintf( "VsyncOn-%s" , name)),
          mDispSync(dispSync),
          mPhaseOffset(base::StringPrintf( "VsyncOffset-%s", name), phaseOffset)    //Initialize
mPhaseOffset When the vsync signal arrives, the offset of sf and app
Copy code
```

Great — you've traced a **complete path from `main()` to `SurfaceFlinger::init()`** and then deep into **HWComposer, RenderEngine, EventThread, and Scheduler setup**. Let's now break this entire process down into **clear, simple explanations** step-by-step. 🧠✨

🏁 1. SurfaceFlinger Starts Up

```cpp
sp<SurfaceFlinger> flinger = surfaceflinger::createSurfaceFlinger();
```

- Calls **`new SurfaceFlinger()`**
- Because **`SurfaceFlinger`** inherits **`RefBase`**, its **`onFirstRef()`** gets called automatically.

🔧 2. `onFirstRef()` → Initialize Event Queue

```cpp
void SurfaceFlinger::onFirstRef() {
    mEventQueue.init(this);
}
```

➤ What this does:

- **Creates a `Looper`**: for handling messages on the main thread.
- **Creates a `Handler`**: to post and process messages related to VSync or transactions.

```cpp
void MessageQueue::init(const sp<SurfaceFlinger>& flinger) {
    mFlinger = flinger;
    mLooper = new Looper(true);      // Prepares a message loop
    mHandler = new Handler(*this);   // Prepares a handler for that loop
}
```

🖼️ 3. `SurfaceFlinger::init()` — Main Initialization

This sets up all the **core graphics hardware/software components**.

🧱 Key Steps:

✅ RenderEngine Creation

```cpp
mCompositionEngine->setRenderEngine(RenderEngine::create(...));
```

- Talks directly to **OpenGL / Vulkan drivers**
- Used for rendering all frames and composition

✅ HWComposer Creation

```cpp
mCompositionEngine->setHwComposer(getFactory().createHWComposer());
```

- Talks to **hardware composer HAL (HWC)** — sends buffers for display
- Receives display events like hotplug

✅ Handle Display Events

```cpp
processDisplayHotplugEventsLocked();
```

- Detects connected displays (e.g., internal screen, HDMI)
- Initializes a default display device

🧠 4. Scheduler and VSync Configuration

```cpp
initScheduler(displayId);
```

This sets up the **frame timing system**: when to draw, how often, how to sync with the display.

⚙️ What Happens Inside:

➤ Create RefreshRateConfigs

```cpp
mRefreshRateConfigs = new RefreshRateConfigs(...);
```

- Manages available display refresh rates (60Hz, 90Hz, etc.)

➤ Create `Scheduler`

```cpp
mScheduler = getFactory().createScheduler(...);
```

- Controls frame pacing (when SF and apps draw)

- Uses VSync signals to maintain consistent timing

➤ Setup Connections

```cpp
mAppConnectionHandle = mScheduler->createConnection("app", ...);
mSfConnectionHandle = mScheduler->createConnection("sf", ...);
```

- **app**: App layer (SurfaceView, OpenGL)
- **sf**: SurfaceFlinger internal work

Each one gets a **VSync source**, a **phase offset**, and an **event thread**.

➤ `DispSyncSource`

```cpp
DispSyncSource::DispSyncSource(DispSync*, phaseOffset, true, name);
```

- Wraps a VSync signal with extra settings like **when to fire callbacks**
- The **offset** allows sf and app to draw at slightly different times (prevent jank)
- On **Systrace**, these appear as **`VSYNC-app`**, **`VSYNC-sf`** markers.

🧵 5. Threads and Message Loops

- **EventThread**: dispatches VSync to app or SF
- **RegionSamplingThread**: reads visible regions for blur/shaders
- **StartPropertySetThread**: sets display boot properties asynchronously

---

🚀 Final Boot Steps

- Draw first frame (boot animation)
- Prime shader cache
- Register Binder service: **`addService("SurfaceFlinger", ...)`**
- Start main run loop: **`flinger->run()`**

🔚 Summary Diagram (Simple View)

```scss
main()
 └─ createSurfaceFlinger()
      ├─ onFirstRef()
      │    └─ Create Looper + Handler
      └─ init()
           ├─ Create RenderEngine
           ├─ Create HWComposer
           ├─ Setup Displays
           ├─ initScheduler()
           │    ├─ Scheduler
           │    ├─ VSync sources (app + sf)
           │    └─ EventThreads
           ├─ Shader cache
           └─ StartPropertySetThread
```

Let me know if you want to walk through how the **app and SurfaceFlinger use VSync differently**, or what **EventThread** does in more detail.

↳

* Draw first frame (boot animation) * Prime shader cache * Register Binder service: `addService("SurfaceFlinger", ...)` * Start main run loop: `flinger->run()`
explain in details

Let's break down those final steps of SurfaceFlinger initialization **in complete detail**, covering:

1. ✅ **Drawing the first frame (boot animation)**
2. 🚀 **Priming the shader cache**
3. 📎 **Registering the SurfaceFlinger Binder service**
4. 🔁 **Starting the main run loop**

✅ 1. Drawing the First Frame (Boot Animation)

🔹 Where it happens:

```cpp
void SurfaceFlinger::init() {
    ...
    startBootAnim();
}
```

🔹 Why it's needed:

- When Android starts up, the **boot animation (bootanimation.zip)** is shown by an app called **bootanimation**.
- But before that, SurfaceFlinger needs to **draw something once**, or the screen might stay black.

🔹 How it works:

```cpp
void SurfaceFlinger::startBootAnim() {
    property_set("service.bootanim.exit", "0");     // Let the bootanimation app run
    property_set("ctl.start", "bootanim");          // Start bootanimation service via init
}
```

- **`bootanimation`** is a small graphics app that uses SurfaceFlinger to draw to the screen.
- SurfaceFlinger **must draw at least one black frame** or **`bootanim`** may not render correctly.
- SurfaceFlinger posts this first frame using **`repaintEverything()`** or **`commitTransactions()`** internally, depending on mode.

🚀 2. Priming the Shader Cache

🔹 Where it happens:

```cpp
void SurfaceFlinger::primeShaderCache() {
    ...
    renderengine->primeCache();
}
```

🔹 Why it's needed:

- When SurfaceFlinger uses **RenderEngine**, it uses **OpenGL/Vulkan shaders** for:
  - Blending
  - Blur
  - Rounded corners
  - Color correction, etc.
- Compiling shaders at runtime causes **jank or stalls**.
- So Android **pre-compiles and caches shaders** used by SurfaceFlinger.

🔹 How it works:

- RenderEngine has a **priming phase**:

```cpp
void RenderEngine::primeCache() {
    // Draw dummy layers using all common shaders
}
```

- This ensures **GPU driver compiles and links** the shaders.
- Next time SF draws, everything is already in memory → no runtime delay.

📎 3. Registering Binder Service

🔹 Where it happens:

```cpp
SurfaceFlinger::init() {
    ...
    defaultServiceManager()->addService(String16("SurfaceFlinger"), this);
}
```

🔹 Why it's important:

- Apps and system services (like WindowManager) talk to SurfaceFlinger using **Binder IPC**.
- SurfaceFlinger needs to expose itself as a **system-wide Binder service**.

🔹 How it works:

- `defaultServiceManager()` gets the Binder system service manager.
- `addService("SurfaceFlinger", this)` registers SF with that name.
- Clients (like `WindowManagerService`) call:

```java
IBinder flinger = ServiceManager.getService("SurfaceFlinger");
```

- Then they call methods like `createSurfaceControl()` or `setTransactionState()`.

  Without this step, no client could connect to SF — no windows, no rendering, nothing.

🔁 4. Start Main Run Loop

🔹 Where it happens:

```cpp
flinger->run();
```

🔹 What it does:

- Enters a **forever loop**, waiting for events (like VSync, transaction commits, or display changes).
- This is powered by the **Looper** created in **`onFirstRef()`**.

🔹 Internals:

```cpp
void SurfaceFlinger::run() {
    do {
        mLooper->pollOnce(-1);  // Block until there's something to do
    } while (!mShuttingDown);
}
```

➤ What kind of events does it wait for?

- **VSync callbacks** → do composition
- **Surface/Layer transactions** → apply new geometry/visibility
- **Hotplug events** → display connected/disconnected
- **MessageQueue events** → handle posted messages from apps or SF itself

This loop keeps SF alive and responsive **until the system shuts down**.

📘 Full Summary Table

Step	Purpose	Code	Impact
✅ Draw first frame	Avoid black screen, kick off boot animation	**`startBootAnim()`**	Boot animation visible via SurfaceFlinger
🚀 Prime shader cache	Precompile shaders to avoid jank	**`primeShaderCache()`**	Smooth future rendering
📎 Register Binder	Allow clients (e.g. WindowManager) to talk to SF	**`addService("SurfaceFlinger", ...)`**	Enables all window creation and composition logic
🔁 Start main run loop	Enter the poll loop for events and VSync	**`flinger->run()`**	Keeps SF alive and active for system lifecycle

---

Would you like to now dive into what exactly happens **inside that main loop** when a frame is drawn? We can walk through **VSync** → **surface draw** → **composition** → **HWComposer** → **display** step-by-step next.

↳

Prime shader
what is this

The term **"Prime shader"** is not a standard or widely used concept in graphics programming (OpenGL, Vulkan, DirectX, etc.), Android, or shader languages (GLSL/HLSL). So, it could mean one of the following depending on **context**:

✅ 1. Custom Project Term or Nickname

- It might be a **project-specific term** used internally by a graphics engine, team, or OEM vendor.
- Example: Someone might call their "main rendering shader" or "important visual effect shader" a *Prime Shader* informally to imply it's the *most crucial* shader in the pipeline.

✅ 2. Misunderstanding or Typo

- It could be a **typo or misinterpretation** of:
  - **"Primary Shader"** (i.e., main fragment/vertex shader)
  - **"Primitive Shader"** (specific to AMD GPU architecture)

✅ 3. If Meant as "Primitive Shader" (AMD)

If you were reading AMD GPU documentation or graphics architecture notes, "Primitive Shader" is a real concept:

📌 Primitive Shader (AMD GPUs)

- Used in **AMD's GCN/RDNA architecture**.
- A GPU stage that **combines vertex and geometry shader stages** for better performance.
- It processes:
  - Vertex shading
  - Culling

    - Primitive assembly
- Purpose: **Improves performance** by reducing fixed-function stages.

But this is **only used in low-level GPU hardware** context, not Android APIs or OpenGL ES.

✅ 4. Main Shader in a Pipeline

- Sometimes "Prime shader" is used to informally refer to the **most important shader** in a graphics pipeline.
    - Example: In a deferred renderer, someone might refer to the G-buffer shader or final composite shader as the "prime shader."

Conclusion:

> **There's no official or standard "Prime Shader" in OpenGL, Vulkan, or Android graphics.**
> If you saw this term in some code or documentation, please share the exact context — then I can give a precise answer.

Would you like to paste where you saw "Prime shader"? I'll explain what it means **in that specific case**.

↳
`VSYNC-app`, `VSYNC-sf`
what are these

The terms **`VSYNC-app`** and **`VSYNC-sf`** refer to **two different VSync timelines** used by the **Android graphics pipeline**.

These are part of how Android coordinates screen refreshes with app rendering and SurfaceFlinger (the system compositor). Let's break this down simply:

🧠 What is VSync?

VSync (Vertical Sync) is a signal from the display hardware that tells the system:

> "I'm about to refresh the screen — if you have a new frame, now's the time to show it!"

In Android, the display usually runs at **60Hz or 120Hz**, so VSync fires every **16.6 ms or 8.3 ms**.

📌 `VSYNC-app`

- This is the **VSync signal sent to the app** process.
- It tells your **application** (like a game or UI) when to start drawing its next frame.
- Used by **Choreographer** and **RenderThread** to kick off rendering.

🟢 Think of it as:

> “Hey App, prepare your next frame now so it’s ready by the next screen refresh.”

📌 `VSYNC-sf` (SurfaceFlinger)

- This is the **VSync signal sent to SurfaceFlinger**, Android’s system compositor.
- SurfaceFlinger **collects all the app frames and composes** them into a final image to send to the display.
- `sf` stands for **SurfaceFlinger**.

🟢 Think of it as:

> “Hey SurfaceFlinger, now it’s your turn to compose the app frames and send the final image to the display.”

🧩 Why Two Different VSyncs?

Because apps and SurfaceFlinger do different jobs and need different **timing**:

- The app gets an earlier signal so it has time to draw its frame.
- SurfaceFlinger gets a slightly later signal, **just in time to composite** all the app frames and push to the screen.

This **prevents frame drops**, avoids tearing, and keeps animation smooth.

🖼️ Timeline (simplified view):

```text
|----- VSYNC-app -----|--- app draws frame --->|
                          |---- VSYNC-sf ----|--- SurfaceFlinger composes ---> display
```

📊 Where You See These?

You’ll see **`VSYNC-app`** and **`VSYNC-sf`** in:

- `dumpsys SurfaceFlinger` **output**
- **systrace / Perfetto timeline**
- Android **FrameTimeline** logs

They help developers debug **frame delays, jank, and latency**.

🔁 Summary

Term	Stands For	Who Uses It	Purpose
`VSYNC-app`	VSync for App	App / Choreographer	Tell app when to start rendering
`VSYNC-sf`	VSync for SurfaceFlinger	SurfaceFlinger	Tell SF when to start composing

---

Would you like a visual timeline or real-world example using systrace?

Create an EventThread object and pass in the vsyncSource object associated with sf or app:

auto eventThread = std::make_unique<impl::EventThread>(std::move(vsyncSource), std::move(interceptCallback));
illustrate:

1) Each ConnectionHandle object has an id, which is used as the key value of the mConnections property (map<id, Connection>) in the Scheduler object. The Connection object contains three properties: ConnectionHandle, EventThreadConnection, and EventThread.

2) In mScheduler->getEventConnection(mSfConnectionHandle), use the id of mSfConnectionHandle as the key value to find the corresponding Connection in Scheduler's mConnections (unordered_map<int64_t, Connection>) and return its EventThreadConnection member property.

3) In getHwComposer().registerCallback(), the registerCallback() methods of HwComposer and Device are called in sequence, and the SurfaceFlinger object is encapsulated into ComposerCallbackBridge in Device; for the encapsulated object, the registerCallback() methods of Composer and IComposerClient are called in sequence to inject it into the implementation class of IComposerClient.

Related questions:

① The screen refresh rate is faster than the system frame rate:

At this time, after the contents of the front buffer are fully mapped to the screen, the back buffer is not yet ready for the next frame. The screen will not be able to read the next frame, so it can only continue to display the graphics of the current frame, causing a frame to be displayed multiple times, which is called stuttering.

② The system frame rate is faster than the screen refresh rate

. At this time, the screen has not completely mapped a frame from the front buffer to the screen, while the system has already prepared the next frame in the back buffer and requires the next frame to be read to the screen. This will cause the upper half of the screen to be the graphics of the previous frame, and the lower half to be the graphics of the next frame, causing multiple frames to be displayed on the screen, which is called screen tearing.

To solve the above problems, the Android display system generally has multiple levels of buffering, that is, while the screen is being refreshed, another buffer is used to prepare the next frame of data, thereby improving performance:

This explanation is about how Android handles **screen refresh synchronization** and how `SurfaceFlinger`, `EventThread`, and `VSync` coordination help manage smooth graphics rendering. Let's go through it piece by piece in **simple terms**.

🔹 What is `EventThread`?

`EventThread` is a class in Android's **SurfaceFlinger** that deals with **vsync events**.

- **VSync (Vertical Sync)** is a signal from the display hardware that tells the system: "Hey, I'm ready for the next frame."
- The `EventThread` receives these VSync signals and **notifies apps or SurfaceFlinger** to start rendering the next frame.
- The line:

  ```cpp
  auto eventThread = std::make_unique<impl::EventThread>(std::move(vsyncSource),
  std::move(interceptCallback));
  ```

  means: create a new `EventThread` object that listens to vsync signals from a `vsyncSource`.

🔹 How connections are managed

1. `ConnectionHandle` and `Scheduler`

- `ConnectionHandle` is like a **ticket or ID** that uniquely identifies a listener (like SurfaceFlinger or an app).

- These are stored in a map:

  ```cpp
  mConnections: map<id, Connection>
  ```

- Each **`Connection`** has:
  - **`ConnectionHandle`** – the ID or key.
  - **`EventThreadConnection`** – the link between client and **`EventThread`**.
  - **`EventThread`** – the object that actually sends vsync signals.

---

2. How SurfaceFlinger gets a vsync connection

```cpp
mScheduler->getEventConnection(mSfConnectionHandle)
```

- This asks the scheduler: **"Give me the EventThreadConnection that belongs to SurfaceFlinger"**
- It finds this by looking up the **`id`** in **`mConnections`**, and returns the **`EventThreadConnection`**.

---

3. Registering for VSync via HWC

```cpp
getHwComposer().registerCallback()
```

- **`HwComposer`** (Hardware Composer HAL) is responsible for talking to the **actual display hardware**.
- When this method is called:
  - SurfaceFlinger wraps itself into a **`ComposerCallbackBridge`**.
  - This bridge is registered with the **`IComposerClient`** (which talks to HWC).
  - So now, when the screen's vsync event happens, HWC tells SurfaceFlinger through this callback.

🔸 Related Problems in Displaying Frames

① Screen refresh rate is higher than frame generation rate (i.e., screen is too fast)

- Display wants new frames quickly, but the app/system hasn't prepared them yet.
- Result: Same frame gets shown again → **Stuttering**.

② Frame generation rate is higher than screen refresh (i.e., system is too fast)

- App/system pushes new frame before screen is done with current one.
- Result: Screen shows part of two frames at once → **Screen tearing** (e.g., top half one frame, bottom half another).

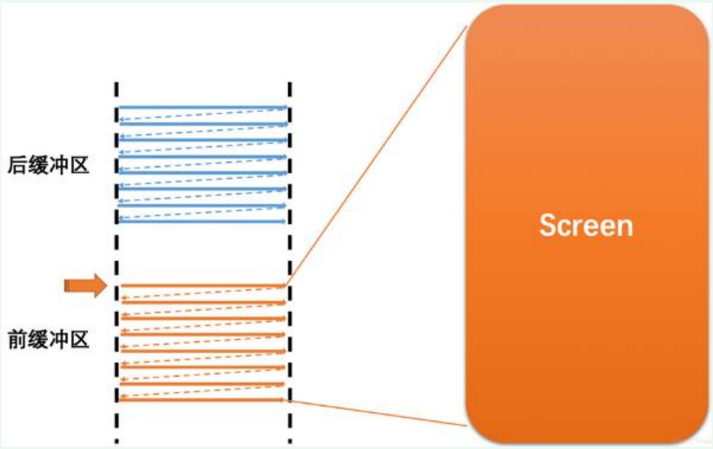
✅ How Android solves this: Buffering

- Uses **multiple buffers**:
  - **Front buffer**: What screen is displaying now.
  - **Back buffer**: What app is drawing next.
  - **(Sometimes extra buffers too)**
- While screen is using one buffer to display a frame, another buffer is used to **prepare the next frame**.
- This is called **double buffering** or **triple buffering**.
- Helps make the rendering **smooth** and avoids tearing or stuttering.

🔁 Summary

Component	Purpose
`EventThread`	Listens to VSync and sends events to apps/SurfaceFlinger
`ConnectionHandle`	Unique ID for a vsync listener
`Scheduler`	Manages all vsync connections
`registerCallback()`	Registers SurfaceFlinger to receive vsync from HWC
Multiple Buffers	Prevent screen tearing or stuttering

Would you like a diagram for this whole flow or want me to break down a particular part in code?



Front buffer: frame buffer used to display content to the screen
Back buffer: frame buffer used to synthesize the next frame of graphics in the background

Vertical synchronization (VSync): A synchronization signal sent after the screen scans a frame from the buffer to the screen and before starting to scan the next frame. This signal is used to switch the front buffer and the back buffer.

Screen refresh rate (HZ): represents the number of times the screen refreshes in one second. Android phones generally refresh at 60HZ (that is, 60 frames per second, and 1 frame in about 16.67 milliseconds).
System frame rate (FPS): represents the number of frames synthesized by the system in one second. The size of this value is determined by the system algorithm and hardware.

This diagram illustrates the **double buffering** mechanism used in graphics rendering, especially relevant to **Android** and other modern systems. Let's break it down step by step with explanation of each part:

🖼️ Image Explanation

On the **left** side of the image:

- The top half labeled **"后缓冲区" (Back Buffer Region)** in blue.
- The bottom half labeled **"前缓冲区" (Front Buffer Region)** in orange.
- Arrows show horizontal scanning of lines — this simulates how a screen renders pixel rows.
- The **orange arrow** in the middle pointing right represents the **VSync signal** (Vertical Sync).
- The **right side**, a rectangular screen, shows that the **front buffer is currently being displayed**.

📘 Terms and Concepts Explained

Term	Meaning
Front Buffer (前缓冲区)	This is the frame currently being shown on the screen.
Back Buffer (后缓冲区)	This is the memory area where the **next frame** is being drawn in the background. It's not visible to the user yet.
VSync (Vertical Synchronization)	A signal that tells the system “the screen has finished drawing the current frame.” This is the **right time to switch** buffers (back → front) to avoid visual glitches like tearing.
Refresh Rate (Hz)	Number of times the screen is refreshed per second (e.g., 60Hz = 60 times per second).
Frame Rate (FPS)	Number of frames the system is **able to draw** per second. This depends on CPU/GPU load and rendering complexity.

🌀 Why This Matters?

Without this system:

- You might see **tearing** — part of a new frame gets shown before the old one is fully removed.
- Or **flickering**, if the frame is drawn while it's being shown.

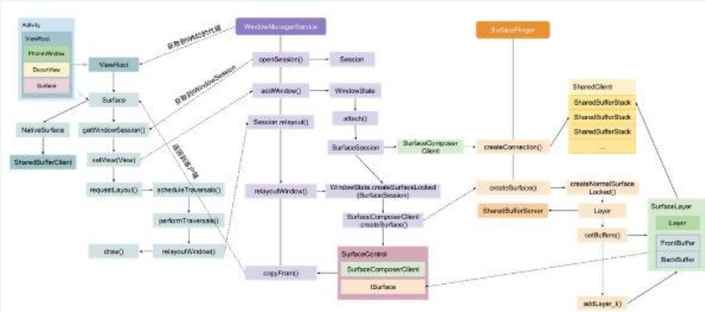
So this buffering + VSync mechanism ensures:

- Smooth animation
- Glitch-free rendering
- Synchronization between GPU-rendering and screen refresh

🔁 The Flow

1. The **GPU** draws the next frame into the **back buffer**.
2. The **screen** reads data from the **front buffer** line-by-line.
3. After the screen is done (VSync triggered), the buffers **swap**.
4. The newly drawn frame in the back buffer becomes the new front buffer.
5. The GPU starts working on the next frame in the now-empty back buffer.
6. Repeat this every 16.67ms (for 60Hz display).

If you'd like, I can also show how **triple buffering**, **adaptive refresh**, or **frame pacing** fit into this system.



4. Surface creation process

The diagram you've uploaded shows the **Surface creation flow** in Android — how an app creates a **`Surface`** to draw on, and how that **`Surface`** is eventually connected to **SurfaceFlinger**, Android's system compositor.

Let me break this down step by step, in **simple and detailed terms**:

🟦 1. App UI Setup

- An **`Activity`** creates UI components like **`ViewRoot`**, **`PhoneWindow`**, **`DecorView`**, etc.
- These are part of the Android **view hierarchy**.
- The **bottom-most component** here is the **`Surface`**.

🟩 2. Surface Creation Request (App Side)

- **`ViewRootImpl`** (shortened as **`ViewRoot`**) is the **bridge between app UI and the system**.
- It calls **`getWindowSession()`** to get a connection to the **WindowManagerService (WMS)**.
- Then it uses **`setView(View)`** to attach the app's top-level view.

From there:

- **`requestLayout()`** → schedules a traversal.
- **`performTraversals()`** → performs layout, measure, and surface updates.
- It calls **`relayoutWindow()`** to tell WMS to **create or resize a Surface**.

🟪 3. WindowManagerService Side

- **`WindowManagerService`** handles windows and surfaces at the system level.
- **`openSession()`** → creates a **`Session`** object.
- **`addWindow()`** → adds the app's window to WMS.

- **`Session.relayout()`** → triggers **`relayoutWindow()`** on the server side.

Inside this, we go deeper:

- **`WindowState.createSurfaceLocked(SurfaceSession)`** is called.
- It uses **`SurfaceComposerClient`** to create a surface.

This is the point where Android starts talking to **SurfaceFlinger**.

🟧 4. SurfaceComposerClient to SurfaceFlinger

- **`SurfaceComposerClient.createSurface()`** connects to SurfaceFlinger and requests a new surface.
- Internally, SurfaceFlinger:
  - **`createConnection()`** → sets up a connection.
  - **`createSurface()`** → allocates a new surface.
  - A **`SharedBufferServer`** is created to manage buffer queues.

🟨 5. Shared Buffer Stack Setup

- SurfaceFlinger sets up:
  - **`SharedClient`**
  - **`SharedBufferStack`**
  - These stacks are buffer queues used for rendering.

A **`Layer`** is also created:

- **`createNormalSurfaceLocked()`** → allocates a **`Layer`**.
- **`setBuffers()`** → sets up front and back buffers.
- **`addLayer_l()`** → adds the layer to the SurfaceFlinger composition list.

🟥 6. Return Surface to App

- The app finally receives a **`SurfaceControl`** object.
- **`SurfaceControl`** wraps an **`ISurface`** interface and gives the app **low-level control** over the surface (position, visibility, buffer submission, etc.).

Now the app can:

- Use a **`Canvas`** or **`OpenGL`** to draw on the surface.

- The drawn frames are sent to SurfaceFlinger for final composition and display.

🔁 Summary (Simplified Flow):

```rust
App -> ViewRootImpl
    -> WindowManagerService (WMS)
        -> SurfaceComposerClient
            -> SurfaceFlinger
                -> Surface/Layer/BufferStack setup
                    -> Return Surface to App
```

Would you like a text version of the flow with just the important method calls, or a side-by-side comparison of roles between the App, WMS, and SurfaceFlinger?

confusing explain in simple words

Let's simplify this step-by-step.

✅ Goal: How does an Android app get a Surface to draw its screen?

A **`Surface`** is like a **blank canvas** where the app draws its screen. That canvas is later shown on the display by **SurfaceFlinger**, Android's system compositor.

🧱 Step 1: App asks for a window

When your app starts:

- Android creates a **window** for it using the **`WindowManager`**.
- That window needs a **surface** (a place to draw).

🖼️ Step 2: App asks for a Surface

The app (via **`ViewRootImpl`**) tells the system:

> "Please give me a surface so I can draw my UI."

This is done using a system call named **`relayoutWindow()`**.

🧠 Step 3: System (WindowManagerService) talks to SurfaceFlinger

The system's `WindowManagerService`:

- Calls `SurfaceComposerClient`, which talks to `SurfaceFlinger`.

Think of it like this:

> "Hey SurfaceFlinger, I need a new surface for this app's window."

🏭 Step 4: SurfaceFlinger creates a Layer (like a sheet of glass)

SurfaceFlinger:

- Creates a **layer**, which is just a visual object to show on screen.
- Sets up **buffers** (memory areas to draw into).
- Gives back a **SurfaceControl**, which the app can use to control the surface.

🎨 Step 5: App uses the Surface to draw

Now the app:

- Draws on the surface using **Canvas**, **OpenGL**, or **Vulkan**.
- When it's done, the surface content is sent to **SurfaceFlinger**.

🖥️ Step 6: SurfaceFlinger puts the surface on screen

SurfaceFlinger takes all surfaces (app, status bar, keyboard, etc.) and **composes** them into the final image.
It sends that image to the **display**.

🧠 Simple Analogy

Component	Like in real life...
App	A painter
Surface	A canvas to draw on
WindowManagerService	The gallery manager who gives canvases to artists
SurfaceFlinger	The final framer who puts all canvases on the wall (display)
SurfaceControl	The handles on the canvas (lets you move or resize it)

Let me know if you'd like a visual drawing to go with this.

The process of creating a Surface is the process of displaying the Activity. The specific implementation of Activity.makeVisible() is called in ActivityThread.handleResumeActivity():

Copy code
```
void makeVisible() {
     if (! mWindowAdded) {
        ViewManager wm = getWindowManager(); // Here getWindowManager gets the
WindowManagerImpl object
        wm.addView(mDecor, getWindow().getAttributes());
        mWindowAdded = true ;
    }
    mDecor.setVisibility(View.VISIBLE);
}
```
Copy code
```
    WindowManagerImpl.java:

public  void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params ) {
    applyDefaultToken( params );
    mGlobal.addView(view, params , mDisplay, mParentWindow);
}
    WindowManagerGlobal.java:
```

Copy code
```
public  void addView(View view, ViewGroup.LayoutParams params , Display display, Window
parentWindow) {
    ...
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params ;
     // Create ViewRootImpl
    ViewRootImpl root = new ViewRootImpl(view.getContext(), display);
    view.setLayoutParams(wparams);
    mViews.add(view);
    mRoots.add(root);
    mParams.add(wparams);
    // Set View
    root.setView(view, wparams, panelParentView);
    ...
}
```
Copy code
```
    Create ViewRootImpl:

    public ViewRootImpl(Context context, Display display) {
```

```
        // Get the proxy class of IWindowSession
        this (context, display, WindowManagerGlobal.getWindowSession(),
                false  /* useSfChoreographer */ );
    }
    WindowManagerGlobal.java:
```

Copy code

```
    @UnsupportedAppUsage
     public  static IWindowSession getWindowSession() {
        synchronized (WindowManagerGlobal. class ) {
            if (sWindowSession == null ) {
                try {
                    // Emulate the legacy behavior. The global instance of InputMethodManager
                    // was instantiated here.
                    // TODO(b/116157766): Remove this hack after cleaning up
@UnsupportedAppUsage
                    // Get IMS The proxy class
                    InputMethodManager.ensureDefaultInstanceForDefaultDisplayIfNecessary();
                    IWindowManager windowManager = getWindowManagerService();
                    // After calling Binder, finally call WMS
                    sWindowSession = windowManager.openSession(
                            new IWindowSessionCallback.Stub() {
                                @Override
                                public  void onAnimatorScaleChanged( float scale) {
                                    ValueAnimator.setDurationScale(scale);
                                }
                            });
                } catch (RemoteException e) {
                    throw e.rethrowFromSystemServer();
                }
            }
            return sWindowSession;
        }
    }
```

Copy code

```
    WindowManagerService.openSession:
```

Copy code

```
    // -------------------------------------------------------------
     // IWindowManager API
     // -------------------------------------------------------------

    @Override
    public IWindowSession openSession(IWindowSessionCallback callback) {
         // Create session object
        return  new Session( this , callback);
```

}
Copy code
The data is written back to the app process through Binder again, and the obtained object is the Session proxy object IWindowSession.
After creating the ViewRootImpl object, call the setView() method of the object:
ViewRootImpl:

Copy code
```
public  void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
  synchronized ( this ) {

    requestLayout(); // See the analysis below for details
    ...
     // Call through Binder and enter the Session of the system process
            res = mWindowSession.addToDisplayAsUser(mWindow, mSeq, mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(), userId, mTmpFrame,
                        mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                        mAttachInfo.mDisplayCutout, inputChannel,
                        mTempInsets, mTempControls);

    ...
  }
}
```
Copy code
Session.java

Copy code
```
    @Override
     public  int addToDisplayAsUser(IWindow window, int seq, WindowManager.LayoutParams
attrs,
            int viewVisibility, int displayId, int userId, Rect outFrame,
            Rect outContentInsets, Rect outStableInsets,
            DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
            InsetsState outInsetsState, InsetsSourceControl[] outActiveControls) {
        // Call the addWindow method of WMS
        return mService.addWindow( this , window, seq, attrs, viewVisibility, displayId, outFrame,
                outContentInsets, outStableInsets, outDisplayCutout, outInputChannel,
                outInsetsState, outActiveControls, userId);
    }
```
Copy code
WindowManagerService.java:

Copy code
```
    public  int addWindow(Session session, IWindow client, int seq,
            LayoutParams attrs, int viewVisibility, int displayId, Rect outFrame,
```

```
            Rect outContentInsets, Rect outStableInsets,
            DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
            InsetsState outInsetsState, InsetsSourceControl[] outActiveControls,
            int requestUserId) {
        Arrays.fill(outActiveControls, null );
         int [] appOp = new  int [1 ];
         final  boolean isRoundedCornerOverlay = (attrs.privateFlags
                & PRIVATE_FLAG_IS_ROUNDED_CORNERS_OVERLAY) != 0 ;
         int res = mPolicy.checkAddPermission(attrs.type, isRoundedCornerOverlay,
attrs.packageName,
                appOp);
        if (res != WindowManagerGlobal.ADD_OKAY) {
            return res;
        }

        WindowState parentWindow = null ;
         final  int callingUid = Binder.getCallingUid();
         final  int callingPid = Binder.getCallingPid();
         final  long origId = Binder.clearCallingIdentity();
         final  int type = attrs.type;

        synchronized (mGlobalLock) {
            if (! mDisplayReady) {
                throw  new IllegalStateException("Display has not been initialized" );
            }
...
            // Create WindowState
            final WindowState win = new WindowState( this , session, client, token, parentWindow,
                    appOp[ 0 ], seq, attrs, viewVisibility, session.mUid, userId,
                    session.mCanAddInternalSystemWindow);
            if (win.mDeathRecipient == null ) {
                // Client has apparently died, so there is no reason to
                // continue.
                ProtoLog.w(WM_ERROR, "Adding window client %s"
                        + " that is dead, aborting." , client.asBinder());
                return WindowManagerGlobal.ADD_APP_EXITING;
            }

            if (win.getDisplayContent() == null ) {
                ProtoLog.w(WM_ERROR, "Adding window to Display that has been removed." );
                return WindowManagerGlobal.ADD_INVALID_DISPLAY;
            }

            // Adjust WindowManager's LayoutParams parameters
            final DisplayPolicy displayPolicy = displayContent.getDisplayPolicy();
            displayPolicy.adjustWindowParamsLw(win, win.mAttrs, callingPid, callingUid);
```

```
            res = displayPolicy.validateAddingWindowLw(attrs, callingPid, callingUid);
             if (res != WindowManagerGlobal.ADD_OKAY) {
                return res;
            }

            // Open the input channel
            final  boolean openInputChannels = (outInputChannel != null
                    && (attrs.inputFeatures & INPUT_FEATURE_NO_INPUT_CHANNEL) == 0 );
             if   (openInputChannels) {
                win.openInputChannel(outInputChannel);
            }

...
            displayContent.getInputMonitor().setUpdateInputWindowsNeededLw();

            boolean focusChanged = false ;
             // When the window can receive key events, update the focused window
            if (win.canReceiveKeys()) {
                focusChanged = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS,
                        false  /* updateInputWindows */ );
                 if (focusChanged) {
                    imMayMove = false ;
                }
            }

            if (imMayMove) {
                displayContent.computeImeTarget( true  /* updateImeTarget */ );
            }
...
        }

        Binder.restoreCallingIdentity(origId);

        return res;
    }
```

Copy code

Create a SurfaceSession object and add the current Session to the WMS.mSessions member variable.

Session.java:

Copy code

```
    void windowAddedLocked(String packageName) {
        mPackageName = packageName;
        mRelayoutTag = "relayoutWindow: " + mPackageName;
         if (mSurfaceSession == null ) {
```

```
            if (DEBUG) {
                Slog.v(TAG_WM, "First window added to " + this + ", creating SurfaceSession" );
            }
            mSurfaceSession = new SurfaceSession();
            ProtoLog.i(WM_SHOW_TRANSACTIONS, " NEW SURFACE SESSION %s" , mSurfaceSession);
            mService.mSessions.add( this );
            if (mLastReportedAnimatorScale != mService.getCurrentAnimatorScale()) {
                mService.dispatchNewAnimatorScaleLocked( this );
            }
        }
        mNumWindow ++ ;
    }
```

Copy code

The creation of SurfaceSession will call JNI, and call nativeCreate() in JNI.

android_view_SurfaceSession.cpp:

```
static jlong nativeCreate(JNIEnv* env, jclass clazz) {
    SurfaceComposerClient * client = new SurfaceComposerClient();
    client ->incStrong(( void * )nativeCreate);
     return reinterpret_cast<jlong> (client);
}
```

Copy code

```
static jlong nativeCreate(JNIEnv* env, jclass clazz, jobject sessionObj,
        jstring nameStr, jint w, jint h, jint format, jint flags, jlong parentObject,
        jobject metadataParcel) {
    ScopedUtfChars name(env, nameStr);
    sp <SurfaceComposerClient> client;
     if (sessionObj != NULL) {
        client = android_view_SurfaceSession_getClient(env, sessionObj);
    } else {
        client = SurfaceComposerClient::getDefault();
    }
    SurfaceControl *parent = reinterpret_cast<SurfaceControl*> (parentObject);
    sp <SurfaceControl> surface;
    LayerMetadata metadata;
    Parcel * parcel = parcelForJavaObject(env, metadataParcel);
     if (parcel && !parcel-> objectsCount()) {
        status_t err = metadata.readFromParcel(parcel);
         if (err != NO_ERROR) {
          jniThrowException(env, "java/lang/IllegalArgumentException" ,
                            "Metadata parcel has wrong format" );
        }
    }

    status_t err = client-> createSurfaceChecked(
            String8(name.c_str()), w, h, format, & surface, flags, parent, std::move(metadata));
```

```
    if (err == NAME_NOT_FOUND) {
       jniThrowException(env, "java/lang/IllegalArgumentException" , NULL);
        return 0 ;
    } else  if (err != NO_ERROR) {
       jniThrowException(env, OutOfResourcesException, NULL);
       return 0 ;
    }

    surface ->incStrong(( void * )nativeCreate);
     return reinterpret_cast<jlong> (surface.get());
}
```
Copy code

Get the SurfaceComposerClient object through the above JNI interface as the proxy object for communicating with SurfaceFlinger.

Copy code
```
void SurfaceComposerClient::onFirstRef() {
     // getComposerService() will return the BpSurfaceFlinger object
    sp<ISurfaceComposer> sf(ComposerService::getComposerService());
     if (sf != nullptr && mStatus == NO_INIT) {
       sp <ISurfaceComposerClient> conn;
        // Call SF's createConnection()
       conn = sf-> createConnection();
        if (conn != nullptr) {
           mClient = conn;
           mStatus = NO_ERROR;
       }
    }
}
```
Copy code

For example, the screenshot interface will call the capture interface through SurfaceControl:

Copy code
```
status_t ScreenshotClient::capture( const sp<IBinder>& display, ui::Dataspace reqDataSpace,
                             ui::PixelFormat reqPixelFormat, const Rect& sourceCrop,
                             uint32_t reqWidth, uint32_t reqHeight, bool useIdentityTransform,
                             ui::Rotation rotation, bool captureSecureLayers,
                             sp <GraphicBuffer>* outBuffer, bool& outCapturedSecureLayers) {
    sp <ISurfaceComposer> s(ComposerService::getComposerService());
     if (s == nullptr) return NO_INIT;
    status_t ret = s-> captureScreen(display, outBuffer, outCapturedSecureLayers, reqDataSpace,
                              reqPixelFormat, sourceCrop, reqWidth, reqHeight,
                              useIdentityTransform, rotation, captureSecureLayers);
    if (ret != NO_ERROR) {
         return ret;
    }
```

```
    return ret;
}
```
Copy code

Then take a closer look at the core SurfaceFlinger implementation:

SurfaceFlinger.cpp:

```
sp<ISurfaceComposerClient> SurfaceFlinger::createConnection() {
     // Create a Client
    const sp<Client> client = new Client( this );
     return client->initCheck() == NO_ERROR ? client : nullptr;
}
```

Back to the previous step, after creating the ViewRootImpl object, the setView() method of the object is called. The requestLayout() method is called in setView(). Now let's take a closer look at the method call process:

Copy code
```
    @Override
     public  void requestLayout() {
        if (! mHandlingLayoutInLayoutRequest) {
            checkThread();
            mLayoutRequested = true ;
            scheduleTraversals();
        }
    }
```
Copy code

Copy code
```
    @UnsupportedAppUsage
     void scheduleTraversals() {
        if (! mTraversalScheduled) {
            mTraversalScheduled = true ;
            mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
                        //Start TraversalRunnable
            mChoreographer.postCallback(
                    Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null );
            notifyRendererOfFramePending();
            pokeDrawLockIfNeeded();
        }
    }
```
Copy code
```
    final  class TraversalRunnable implements Runnable {
        @Override
        public  void run() {
            doTraversal();
        }
    }
```

Copy code

```
void doTraversal() {
     if (mTraversalScheduled) {
        mTraversalScheduled = false ;
        mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

        if (mProfile) {
            Debug.startMethodTracing( "ViewAncestor" );
        }
                     //Call performTraversals
         performTraversals();
if (mProfile) {
            Debug.stopMethodTracing();
            mProfile = false ;
        }
    }
}
```

Copy code

Copy code

```
private  void performTraversals() {
     // cache mView since it is used so much below...
    final View host = mView; // It is DecorView
...
     if (mFirst || windowShouldResize || viewVisibilityChanged || cutoutChanged || params !=
null
            || mForceNextWindowRelayout) {
        mForceNextWindowRelayout = false ;

        if (isViewVisible) {
             // If this window is giving internal insets to the window
             // manager, and it is being added or changing its visibility,
             // then we want to first give the window manager "fake"
             // insets to cause it to effectively ignore the content of
             // the window during layout. This avoids it briefly causing
             // other windows to resize/move based on the raw frame of the
             // window, waiting until we can finish laying out this window
             // and get back to the window manager with the ultimately
             // computed insets.
            insetsPending = computesInternalInsets && (mFirst || viewVisibilityChanged);
        }
...
        try {
             if (DEBUG_LAYOUT) {
                Log.i(mTag, "host=w:" + host.getMeasuredWidth() + ", h:" +
                        host.getMeasuredHeight() + ", params=" + params);
            }
```

```
                if (mAttachInfo.mThreadedRenderer != null ) {
                    // relayoutWindow may decide to destroy mSurface. As that decision
                    // happens in WindowManager service, we need to be defensive here
                    // and stop using the surface in case it gets destroyed.
                    if (mAttachInfo.mThreadedRenderer.pause()) {
                        // Animations were running so we need to push a frame
                        // to resume them
                        mDirty.set(0, 0 , mWidth, mHeight);
                    }

    mChoreographer.mFrameInfo.addFlags(FrameInfo.FLAG_WINDOW_LAYOUT_CHANGED);
                }
                // Key function relayoutWindow
                relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);

                if (DEBUG_LAYOUT) Log.v(mTag, "relayout: frame=" + frame.toShortString()
                        + " cutout=" + mPendingDisplayCutout.get().toString()
                        + " surface=" + mSurface);

                // If the pending {@link MergedConfiguration} handed back from
                 // {@link #relayoutWindow} does not match the one last reported,
                 // WindowManagerService has reported back a frame from a configuration not yet
                 // handled by the client. In this case, we need to accept the configuration so we
                 // do not lay out and draw with the wrong configuration.
                if (! mPendingMergedConfiguration.equals(mLastReportedMergedConfiguration)) {
                    if (DEBUG_CONFIGURATION) Log.v(mTag, "Visible with new config: "
                            + mPendingMergedConfiguration.getMergedConfiguration());
                    performConfigurationChange(mPendingMergedConfiguration, ! mFirst,
                            INVALID_DISPLAY /* same display */ );
                    updatedConfiguration = true ;
                }

            } catch (RemoteException e) {
            }
...
        }

        boolean cancelDraw = mAttachInfo.mTreeObserver.dispatchOnPreDraw() || ! isViewVisible;

        if (! cancelDraw) {
            if (mPendingTransitions != null && mPendingTransitions.size() > 0 ) {
                for ( int i = 0; i < mPendingTransitions.size(); ++ i) {
                    mPendingTransitions.get(i).startChangingAnimations();
                }
                mPendingTransitions.clear();
```

```
            }

            // Start drawing, which calls draw(fullRedrawNeeded);
            performDraw();
        } else {
             if (isViewVisible) {
                 // Try again
                scheduleTraversals();
            } else  if (mPendingTransitions != null && mPendingTransitions.size() > 0 ) {
                 for ( int i = 0; i < mPendingTransitions.size(); ++ i) {
                    mPendingTransitions.get(i).endChangingAnimations();
                }
                mPendingTransitions.clear();
            }
        }

        if (mAttachInfo.mContentCaptureEvents != null ) {
            notifyContentCatpureEvents();
        }

        mIsInTraversal = false ;
    }
```
Copy code

Let's look at the implementation of relayoutWindow:

Copy code

```
...
        if (mSurfaceControl.isValid()) {
             if (! useBLAST()) {
                 // First create a local Surface, then call copyFrom to copy the SurfaceControl
information to Surface
                mSurface.copyFrom(mSurfaceControl);
            } else {
                 final Surface blastSurface = getOrCreateBLASTSurface(mSurfaceSize.x,
                    mSurfaceSize.y);
                // If blastSurface == null that means it hasn't changed since the last time we
                 // called. In this situation, avoid calling transferFrom as we would then
                 // inc the generation ID and cause EGL resources to be recreated.
                if (blastSurface != null ) {
                    mSurface.transferFrom(blastSurface);
                }
            }
        } else {
            destroySurface();
        }
....
```

Copy code
The SurfaceControl class can be considered as a wrapper class, which will eventually execute copyFrom() to return it to the App client:

Copy code

```
@UnsupportedAppUsage
 public  void copyFrom(SurfaceControl other) {
    if (other == null ) {
        throw  new IllegalArgumentException("other must not be null" );
   }

   long surfaceControlPtr = other.mNativeObject;
    if (surfaceControlPtr == 0 ) {
        throw  new NullPointerException(
                "null SurfaceControl native object. Are you using a released SurfaceControl?" );
   }

   // Get the source SurfaceControl through JNI
   long newNativeObject = nativeGetFromSurfaceControl(mNativeObject, surfaceControlPtr);

   synchronized (mLock) {
        if (newNativeObject == mNativeObject) {
            return ;
       }
       if (mNativeObject != 0 ) {
           nativeRelease(mNativeObject);
       }
       // Save to global mNativeObject for external call
       setNativeObjectLocked(newNativeObject);
   }
}
```

Copy code

Thanks for sharing the image. Let's now go over it in **simple terms**, with each part explained like a story:

🧠 What you're looking at:

This image shows how **SurfaceFlinger** works inside Android to draw everything on the screen — from apps, status bar, keyboard, etc.

🧱 Key Layers in the Diagram:

1. App Process (left box)

- This is your Android app (e.g., Facebook, YouTube).
- It wants to show something on the screen.

✍️ **It uses:**

- `Surface` to draw on.
- `SurfaceControl` to talk to **SurfaceFlinger** (to say things like move, hide, resize).

2. SurfaceFlinger Process (middle box)

This is a **system service** whose job is to **collect surfaces from all apps** and **compose** (combine) them into one image to show on the screen.

🛠️ **It handles:**

- Layers: One for each app or UI piece (like status bar).
- Composition: Blending all layers into a final image.
- Hardware Composer (HWC): Sends the final image to display.

3. Hardware Composer (right box)

- This is either **hardware or driver software**.
- Gets the final image and shows it on your display.

🎯 Main Flow (Simple Version):

1. **App creates a surface**
   - App says: “I need a canvas to draw on.”
   - Gets a `Surface` and a `SurfaceControl`.
2. **App draws something**
   - Maybe some buttons, images, text.
   - Draws it using OpenGL, Vulkan, Canvas, etc.
3. **App sends the drawing to SurfaceFlinger**
   - When drawing is done, it’s passed to SurfaceFlinger.
4. **SurfaceFlinger collects all surfaces**
   - From your app
   - From system UI (status bar, keyboard, etc.)
   - From animations

5. **SurfaceFlinger composes them together**
    - Like layers in Photoshop, it puts them in order.
    - It clips, scales, blends them.
6. **Sends final image to Hardware Composer**
    - The hardware shows the final frame on your screen.

🧠 Why SurfaceControl?

- Think of **`SurfaceControl`** as a **remote control** to manage the surface:
    - Move it
    - Resize it
    - Change transparency
    - Stack above/below others

The app uses this to **tell SurfaceFlinger how to treat the surface**, but the app **can't draw directly with it** — drawing is done via **`Surface`**.

📌 Summary (like a team):

Component	Role
App	Creates surface & draws content
Surface	Canvas (where drawing happens)
SurfaceControl	Remote control to tell SurfaceFlinger how to place/show the surface
SurfaceFlinger	The painter-in-chief: collects surfaces and makes a final image
Hardware Composer	The delivery guy: sends final image to screen

---

Let me know if you'd like a **simplified sketch** of this flow.

Surfaceflinger-2

The process of creating a Surface is the process of displaying the Activity. The specific implementation of Activity.makeVisible() is called in ActivityThread.handleResumeActivity():

Copy code
```
void makeVisible() {
     if (! mWindowAdded) {
        ViewManager wm = getWindowManager(); // Here getWindowManager gets the
WindowManagerImpl object
        wm.addView(mDecor, getWindow().getAttributes());
        mWindowAdded = true ;
    }
    mDecor.setVisibility(View.VISIBLE);
}
```
Copy code
WindowManagerImpl.java:

```
public  void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params ) {
    applyDefaultToken( params );
    mGlobal.addView(view, params , mDisplay, mParentWindow);
}
```
WindowManagerGlobal.java:

Copy code
```
public  void addView(View view, ViewGroup.LayoutParams params , Display display, Window
parentWindow) {
    ...
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params ;
     // Create ViewRootImpl
    ViewRootImpl root = new ViewRootImpl(view.getContext(), display);
    view.setLayoutParams(wparams);
    mViews.add(view);
    mRoots.add(root);
    mParams.add(wparams);
    // Set View
    root.setView(view, wparams, panelParentView);
    ...
}
```
Copy code
Create ViewRootImpl:

```
    public ViewRootImpl(Context context, Display display) {
        // Get the proxy class of IWindowSession
        this (context, display, WindowManagerGlobal.getWindowSession(),
                false  /* useSfChoreographer */ );
    }
    WindowManagerGlobal.java:
```

Copy code

```
    @UnsupportedAppUsage
     public  static IWindowSession getWindowSession() {
         synchronized (WindowManagerGlobal. class ) {
             if (sWindowSession == null ) {
                 try {
                     // Emulate the legacy behavior. The global instance of InputMethodManager
                     // was instantiated here.
                     // TODO(b/116157766): Remove this hack after cleaning up
@UnsupportedAppUsage
                     // Get IMS The proxy class
                    InputMethodManager.ensureDefaultInstanceForDefaultDisplayIfNecessary();
                    IWindowManager windowManager = getWindowManagerService();
                     // After calling Binder, finally call WMS
                    sWindowSession = windowManager.openSession(
                            new IWindowSessionCallback.Stub() {
                                @Override
                                public  void onAnimatorScaleChanged( float scale) {
                                    ValueAnimator.setDurationScale(scale);
                                }
                            });
                } catch (RemoteException e) {
                     throw e.rethrowFromSystemServer();
                }
            }
            return sWindowSession;
        }
    }
```

Copy code

```
    WindowManagerService.openSession:
```

Copy code

```
    // -------------------------------------------------------------
     // IWindowManager API
     // -------------------------------------------------------------

    @Override
    public IWindowSession openSession(IWindowSessionCallback callback) {
```

```
     // Create session object
    return  new Session( this , callback);
  }
```
Copy code

The data is written back to the app process through Binder again, and the obtained object is the Session proxy object IWindowSession.

After creating the ViewRootImpl object, call the setView() method of the object:

ViewRootImpl:

Copy code

```
public  void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
  synchronized ( this ) {

    requestLayout(); // See the analysis below for details
    ...
     // Call through Binder and enter the Session of the system process
            res = mWindowSession.addToDisplayAsUser(mWindow, mSeq, mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(), userId, mTmpFrame,
                        mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                        mAttachInfo.mDisplayCutout, inputChannel,
                        mTempInsets, mTempControls);

    ...
  }
}
```
Copy code

Session.java

Copy code

```
    @Override
     public  int addToDisplayAsUser(IWindow window, int seq, WindowManager.LayoutParams
attrs,
            int viewVisibility, int displayId, int userId, Rect outFrame,
            Rect outContentInsets, Rect outStableInsets,
            DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
            InsetsState outInsetsState, InsetsSourceControl[] outActiveControls) {
        // Call the addWindow method of WMS
        return mService.addWindow( this , window, seq, attrs, viewVisibility, displayId, outFrame,
                outContentInsets, outStableInsets, outDisplayCutout, outInputChannel,
                outInsetsState, outActiveControls, userId);
    }
```
Copy code

WindowManagerService.java:

Copy code

```
    public  int addWindow(Session session, IWindow client, int seq,
            LayoutParams attrs, int viewVisibility, int displayId, Rect outFrame,
            Rect outContentInsets, Rect outStableInsets,
            DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
            InsetsState outInsetsState, InsetsSourceControl[] outActiveControls,
            int requestUserId) {
        Arrays.fill(outActiveControls, null );
         int [] appOp = new  int [1 ];
         final  boolean isRoundedCornerOverlay = (attrs.privateFlags
                & PRIVATE_FLAG_IS_ROUNDED_CORNERS_OVERLAY) != 0 ;
         int res = mPolicy.checkAddPermission(attrs.type, isRoundedCornerOverlay,
  attrs.packageName,
                appOp);
        if (res != WindowManagerGlobal.ADD_OKAY) {
             return res;
        }

        WindowState parentWindow = null ;
         final  int callingUid = Binder.getCallingUid();
         final  int callingPid = Binder.getCallingPid();
         final  long origId = Binder.clearCallingIdentity();
         final  int type = attrs.type;

        synchronized (mGlobalLock) {
             if (! mDisplayReady) {
                 throw  new IllegalStateException("Display has not been initialized" );
            }
...
            // Create WindowState
            final WindowState win = new WindowState( this , session, client, token, parentWindow,
                    appOp[ 0 ], seq, attrs, viewVisibility, session.mUid, userId,
                    session.mCanAddInternalSystemWindow);
            if (win.mDeathRecipient == null ) {
                 // Client has apparently died, so there is no reason to
                 // continue.
                ProtoLog.w(WM_ERROR, "Adding window client %s"
                        + " that is dead, aborting." , client.asBinder());
                 return WindowManagerGlobal.ADD_APP_EXITING;
            }

            if (win.getDisplayContent() == null ) {
                ProtoLog.w(WM_ERROR, "Adding window to Display that has been removed." );
                 return WindowManagerGlobal.ADD_INVALID_DISPLAY;
            }

            // Adjust WindowManager's LayoutParams parameters
```

```
            final DisplayPolicy displayPolicy = displayContent.getDisplayPolicy();
            displayPolicy.adjustWindowParamsLw(win, win.mAttrs, callingPid, callingUid);

            res = displayPolicy.validateAddingWindowLw(attrs, callingPid, callingUid);
             if (res != WindowManagerGlobal.ADD_OKAY) {
                 return res;
            }

            // Open the input channel
            final  boolean openInputChannels = (outInputChannel != null
                    && (attrs.inputFeatures & INPUT_FEATURE_NO_INPUT_CHANNEL) == 0 );
             if   (openInputChannels) {
                win.openInputChannel(outInputChannel);
            }


...
            displayContent.getInputMonitor().setUpdateInputWindowsNeededLw();

            boolean focusChanged = false ;
             // When the window can receive key events, update the focused window
            if (win.canReceiveKeys()) {
                focusChanged = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS,
                        false  /* updateInputWindows */ );
                 if (focusChanged) {
                    imMayMove = false ;
                }
            }

            if (imMayMove) {
                displayContent.computeImeTarget( true  /* updateImeTarget */ );
            }
...
        }

        Binder.restoreCallingIdentity(origId);

        return res;
    }
```
Copy code

Create a SurfaceSession object and add the current Session to the WMS.mSessions member variable.

Session.java:

Copy code
```
    void windowAddedLocked(String packageName) {
        mPackageName = packageName;
```

```
        mRelayoutTag = "relayoutWindow: " + mPackageName;
         if (mSurfaceSession == null ) {
             if (DEBUG) {
                Slog.v(TAG_WM, "First window added to " + this + ", creating SurfaceSession" );
            }
            mSurfaceSession = new SurfaceSession();
            ProtoLog.i(WM_SHOW_TRANSACTIONS, " NEW SURFACE SESSION %s" , mSurfaceSession);
            mService.mSessions.add( this );
             if (mLastReportedAnimatorScale != mService.getCurrentAnimatorScale()) {
                mService.dispatchNewAnimatorScaleLocked( this );
            }
        }
        mNumWindow ++ ;
    }
```

Copy code

The creation of SurfaceSession will call JNI, and call nativeCreate() in JNI.

android_view_SurfaceSession.cpp:

```
static jlong nativeCreate(JNIEnv* env, jclass clazz) {
    SurfaceComposerClient * client = new SurfaceComposerClient();
    client ->incStrong(( void * )nativeCreate);
     return reinterpret_cast<jlong> (client);
}
```

Copy code

```
static jlong nativeCreate(JNIEnv* env, jclass clazz, jobject sessionObj,
        jstring nameStr, jint w, jint h, jint format, jint flags, jlong parentObject,
        jobject metadataParcel) {
    ScopedUtfChars name(env, nameStr);
    sp <SurfaceComposerClient> client;
     if (sessionObj != NULL) {
        client = android_view_SurfaceSession_getClient(env, sessionObj);
    } else {
        client = SurfaceComposerClient::getDefault();
    }
    SurfaceControl *parent = reinterpret_cast<SurfaceControl*> (parentObject);
    sp <SurfaceControl> surface;
    LayerMetadata metadata;
    Parcel * parcel = parcelForJavaObject(env, metadataParcel);
     if (parcel && !parcel-> objectsCount()) {
        status_t err = metadata.readFromParcel(parcel);
         if (err != NO_ERROR) {
          jniThrowException(env, "java/lang/IllegalArgumentException" ,
                          "Metadata parcel has wrong format" );
        }
    }
```

```
    status_t err = client-> createSurfaceChecked(
            String8(name.c_str()), w, h, format, & surface, flags, parent, std::move(metadata));
     if (err == NAME_NOT_FOUND) {
        jniThrowException(env, "java/lang/IllegalArgumentException" , NULL);
         return 0 ;
    } else  if (err != NO_ERROR) {
        jniThrowException(env, OutOfResourcesException, NULL);
        return 0 ;
    }

    surface ->incStrong(( void * )nativeCreate);
     return reinterpret_cast<jlong> (surface.get());
}
```

Copy code

Get the SurfaceComposerClient object through the above JNI interface as the proxy object for communicating with SurfaceFlinger.

Copy code

```
void SurfaceComposerClient::onFirstRef() {
     // getComposerService() will return the BpSurfaceFlinger object
    sp<ISurfaceComposer> sf(ComposerService::getComposerService());
     if (sf != nullptr && mStatus == NO_INIT) {
        sp <ISurfaceComposerClient> conn;
         // Call SF's createConnection()
        conn = sf-> createConnection();
         if (conn != nullptr) {
            mClient = conn;
            mStatus = NO_ERROR;
        }
    }
}
```

Copy code

For example, the screenshot interface will call the capture interface through SurfaceControl:

Copy code

```
status_t ScreenshotClient::capture( const sp<IBinder>& display, ui::Dataspace reqDataSpace,
                                    ui::PixelFormat reqPixelFormat, const Rect& sourceCrop,
                                    uint32_t reqWidth, uint32_t reqHeight, bool useIdentityTransform,
                                    ui::Rotation rotation, bool captureSecureLayers,
                                    sp <GraphicBuffer>* outBuffer, bool& outCapturedSecureLayers) {
    sp <ISurfaceComposer> s(ComposerService::getComposerService());
     if (s == nullptr) return NO_INIT;
    status_t ret = s-> captureScreen(display, outBuffer, outCapturedSecureLayers, reqDataSpace,
                                     reqPixelFormat, sourceCrop, reqWidth, reqHeight,
                                     useIdentityTransform, rotation, captureSecureLayers);
    if (ret != NO_ERROR) {
```

```
        return ret;
    }
    return ret;
}
```

Copy code

Then take a closer look at the core SurfaceFlinger implementation:

SurfaceFlinger.cpp:

```
sp<ISurfaceComposerClient> SurfaceFlinger::createConnection() {
     // Create a Client
    const sp<Client> client = new Client( this );
     return client->initCheck() == NO_ERROR ? client : nullptr;
}
```

Back to the previous step, after creating the ViewRootImpl object, the setView() method of the object is called. The requestLayout() method is called in setView(). Now let's take a closer look at the method call process:

Copy code

```
    @Override
     public  void requestLayout() {
        if (! mHandlingLayoutInLayoutRequest) {
            checkThread();
            mLayoutRequested = true ;
            scheduleTraversals();
        }
    }
```

Copy code

Copy code

```
    @UnsupportedAppUsage
     void scheduleTraversals() {
        if (! mTraversalScheduled) {
            mTraversalScheduled = true ;
            mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
                        //Start TraversalRunnable
            mChoreographer.postCallback(
                    Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null );
            notifyRendererOfFramePending();
            pokeDrawLockIfNeeded();
        }
    }
```

Copy code

```
    final  class TraversalRunnable implements Runnable {
        @Override
        public  void run() {
            doTraversal();
```

```
        }
    }
Copy code
    void doTraversal() {
         if (mTraversalScheduled) {
            mTraversalScheduled = false ;
            mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

            if (mProfile) {
                Debug.startMethodTracing( "ViewAncestor" );
            }
                         //Call performTraversals
             performTraversals();
if (mProfile) {
                Debug.stopMethodTracing();
                mProfile = false ;
            }
        }
    }
Copy code
Copy code
    private  void performTraversals() {
         // cache mView since it is used so much below...
        final View host = mView; // It is DecorView
...
         if (mFirst || windowShouldResize || viewVisibilityChanged || cutoutChanged || params !=
null
                || mForceNextWindowRelayout) {
            mForceNextWindowRelayout = false ;

            if (isViewVisible) {
                 // If this window is giving internal insets to the window
                 // manager, and it is being added or changing its visibility,
                 // then we want to first give the window manager "fake"
                 // insets to cause it to effectively ignore the content of
                 // the window during layout. This avoids it briefly causing
                 // other windows to resize/move based on the raw frame of the
                 // window, waiting until we can finish laying out this window
                 // and get back to the window manager with the ultimately
                 // computed insets.
                insetsPending = computesInternalInsets && (mFirst || viewVisibilityChanged);
            }
...
            try {
                 if (DEBUG_LAYOUT) {
                    Log.i(mTag, "host=w:" + host.getMeasuredWidth() + ", h:" +
```

```
                host.getMeasuredHeight() + ", params=" + params);
            }

            if (mAttachInfo.mThreadedRenderer != null ) {
                // relayoutWindow may decide to destroy mSurface. As that decision
                // happens in WindowManager service, we need to be defensive here
                // and stop using the surface in case it gets destroyed.
                if (mAttachInfo.mThreadedRenderer.pause()) {
                    // Animations were running so we need to push a frame
                    // to resume them
                    mDirty.set(0, 0 , mWidth, mHeight);
                }

    mChoreographer.mFrameInfo.addFlags(FrameInfo.FLAG_WINDOW_LAYOUT_CHANGED);
            }
            // Key function relayoutWindow
            relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);

            if (DEBUG_LAYOUT) Log.v(mTag, "relayout: frame=" + frame.toShortString()
                    + " cutout=" + mPendingDisplayCutout.get().toString()
                    + " surface=" + mSurface);

            // If the pending {@link MergedConfiguration} handed back from
            // {@link #relayoutWindow} does not match the one last reported,
            // WindowManagerService has reported back a frame from a configuration not yet
            // handled by the client. In this case, we need to accept the configuration so we
            // do not lay out and draw with the wrong configuration.
            if (! mPendingMergedConfiguration.equals(mLastReportedMergedConfiguration)) {
                if (DEBUG_CONFIGURATION) Log.v(mTag, "Visible with new config: "
                        + mPendingMergedConfiguration.getMergedConfiguration());
                performConfigurationChange(mPendingMergedConfiguration, ! mFirst,
                        INVALID_DISPLAY /* same display */ );
                updatedConfiguration = true ;
            }

        } catch (RemoteException e) {
        }
...
    }

    boolean cancelDraw = mAttachInfo.mTreeObserver.dispatchOnPreDraw() || ! isViewVisible;

    if (! cancelDraw) {
        if (mPendingTransitions != null && mPendingTransitions.size() > 0 ) {
            for ( int i = 0; i < mPendingTransitions.size(); ++ i) {
                mPendingTransitions.get(i).startChangingAnimations();
```

```
            }
            mPendingTransitions.clear();
        }

        // Start drawing, which calls draw(fullRedrawNeeded);
        performDraw();
    } else {
         if (isViewVisible) {
             // Try again
            scheduleTraversals();
        } else  if (mPendingTransitions != null && mPendingTransitions.size() > 0 ) {
             for ( int i = 0; i < mPendingTransitions.size(); ++ i) {
                mPendingTransitions.get(i).endChangingAnimations();
            }
            mPendingTransitions.clear();
        }
    }

    if (mAttachInfo.mContentCaptureEvents != null ) {
        notifyContentCatpureEvents();
    }

    mIsInTraversal = false ;
}
```
Copy code
Let's look at the implementation of relayoutWindow:

Copy code
```
...
    if (mSurfaceControl.isValid()) {
         if (! useBLAST()) {
             // First create a local Surface, then call copyFrom to copy the SurfaceControl
information to Surface
            mSurface.copyFrom(mSurfaceControl);
        } else {
             final Surface blastSurface = getOrCreateBLASTSurface(mSurfaceSize.x,
                mSurfaceSize.y);
            // If blastSurface == null that means it hasn't changed since the last time we
             // called. In this situation, avoid calling transferFrom as we would then
             // inc the generation ID and cause EGL resources to be recreated.
            if (blastSurface != null ) {
                mSurface.transferFrom(blastSurface);
            }
        }
    } else {
        destroySurface();
```

```
        }
    ....
    Copy code
      The SurfaceControl class can be considered as a wrapper class, which will eventually execute
    copyFrom() to return it to the App client:

    Copy code
      @UnsupportedAppUsage
       public  void copyFrom(SurfaceControl other) {
          if (other == null ) {
              throw  new IllegalArgumentException("other must not be null" );
          }

          long surfaceControlPtr = other.mNativeObject;
          if (surfaceControlPtr == 0 ) {
              throw  new NullPointerException(
                      "null SurfaceControl native object. Are you using a released SurfaceControl?" );
          }

          // Get the source SurfaceControl through JNI
          long newNativeObject = nativeGetFromSurfaceControl(mNativeObject, surfaceControlPtr);

          synchronized (mLock) {
              if (newNativeObject == mNativeObject) {
                  return ;
              }
              if (mNativeObject != 0 ) {
                  nativeRelease(mNativeObject);
              }
              // Save to global mNativeObject for external call
              setNativeObjectLocked(newNativeObject);
          }
      }
    Copy code
```

You've traced the entire **Surface creation pipeline** in Android—from the app process to **`WindowManagerService`** and back—beautifully. Let's break this down and explain **each key step** in **simple words**, with a clear focus on the flow of how a **`Surface`** is created and shown when an **`Activity`** becomes visible.

🧱 Summary: What is Happening?

When your Activity is resumed, Android goes through a complex pipeline that ends in creating a **Surface** using **SurfaceFlinger**, assigning it to your window, and letting it draw. Here's a simple picture:

```nginx
Activity → WindowManager → ViewRootImpl → SurfaceSession → SurfaceControl → SurfaceFlinger →
Surface
```

🔁 Step-by-Step Breakdown in Simple Terms

1. Activity Resumes → Calls `makeVisible()`

```java
if (!mWindowAdded) {
    wm.addView(mDecor, getWindow().getAttributes());
}
```

- The **DecorView** (top-level View of your Activity) is added to the **WindowManager**.
- This triggers the whole Surface creation pipeline.

---

2. `WindowManagerGlobal.addView()`

```java
ViewRootImpl root = new ViewRootImpl(view.getContext(), display);
root.setView(view, wparams, panelParentView);
```

- **`ViewRootImpl`** is created. It's a bridge between your app's UI and the windowing system.
- It handles **layout**, **drawing**, **input**, and **Surface management**.

---

3. `ViewRootImpl.setView()` calls `mWindowSession.addToDisplayAsUser()`

- **`mWindowSession`** is a **proxy interface (`IWindowSession`)** which talks to the **WindowManagerService (WMS)** via Binder.
- This tells the system: “I want to add a window and need a surface.”

---

4. WMS receives request: `addWindow()`

```java
final WindowState win = new WindowState(...);
win.openInputChannel(outInputChannel);
```

- WMS creates a **WindowState**, representing your app's window.

- It checks permissions, sets up focus, and prepares input.
- **Now it needs a Surface.**

---

5. Session’s `windowAddedLocked()` → `SurfaceSession` is created

```java
mSurfaceSession = new SurfaceSession();
```

- **`SurfaceSession`** is a low-level object that manages **SurfaceControl** objects.
- This step goes **into JNI**.

---

6. JNI: `nativeCreate()` → creates a `SurfaceComposerClient`

```cpp
SurfaceComposerClient* client = new SurfaceComposerClient();
```

- **`SurfaceComposerClient`** is the interface to **SurfaceFlinger**.
- It’s responsible for **asking SurfaceFlinger to create a new surface**.

---

7. `SurfaceComposerClient::createSurfaceChecked()`

```cpp
sf->createConnection() → new Client() → new SurfaceControl
```

- Now the **real surface** is created by talking to **SurfaceFlinger**.
- Internally, it uses **BufferQueue + GraphicBuffer + SurfaceControl**.

---

8. Back in App: `ViewRootImpl.performTraversals()`

```java
relayoutResult = relayoutWindow(...);
```

- Triggers a **layout/draw cycle** and relayouts the window with the real size and Surface.

---

9. relayoutWindow() → copies Surface from SurfaceControl

```java
mSurface.copyFrom(mSurfaceControl);
```

- Finally, the app's **`Surface`** is filled using **`SurfaceControl`**.
- Now the app can **draw to this surface using Canvas or OpenGL**.

---

10. Draw Frame → Render to Screen

```java
performDraw();
```

- The app renders using **`Canvas`**, **`RenderThread`**, or **`OpenGL`**.
- SurfaceFlinger will later **compose this surface to the display.**

📌 Key Concepts Explained

✅ SurfaceControl

- Think of it as a **handle to a layer** managed by SurfaceFlinger.
- It doesn't draw directly; it **represents metadata** (size, position, z-order).

✅ Surface

- This is the **actual memory buffer** (double/triple buffer) you draw to.
- The app writes pixels to it via **`Canvas`**, **`HardwareRenderer`**, or OpenGL.

✅ SurfaceSession

- A container for managing **multiple SurfaceControls**.
- Created only once per window/session.

✅ SurfaceFlinger

- Android's **system compositor**.
- It gets Surfaces from all apps, composes them into one image, and sends it to the display.

🖼️ Visualization

```scss
Activity.makeVisible()
        ↓
WindowManager.addView()
        ↓
ViewRootImpl created → setView()
        ↓
mWindowSession.addToDisplay() → Binder IPC
        ↓
WMS.addWindow() → creates SurfaceSession
        ↓
SurfaceComposerClient (via JNI)
        ↓
SurfaceControl created by SurfaceFlinger
        ↓
Surface.copyFrom(SurfaceControl)
        ↓
App can now draw → SurfaceFlinger displays it
```

🚀 Summary

- Your **Activity window** ends up as a **Surface** inside **SurfaceFlinger**.
- The **WMS** mediates the window's lifecycle.
- **SurfaceControl** is a controller.
- **Surface** is what your app draws on.
- **SurfaceFlinger** is what shows it on screen.

Let me know if you want the **drawing** side (Choreographer + RenderThread + VSync), or how SurfaceFlinger composes all layers.

yes

Great! Let's now walk through the **drawing side** of the pipeline in Android — how your app actually renders to the screen **after** the Surface has been created.

🧭 Overview: From `requestLayout()` to Pixels on Screen

After the `Surface` is created, the actual **drawing** goes through:

```mathematica
App Thread → ViewRootImpl → Choreographer → RenderThread → SurfaceFlinger → Display
```

🧱 Drawing Pipeline: Step-by-Step

🔹 1. `ViewRootImpl.requestLayout()` triggers traversal

```java
requestLayout(); → scheduleTraversals(); → doTraversal(); → performTraversals();
```

- This is initiated when the UI changes (like `TextView.setText()`).
- Schedules a frame to render using **Choreographer**.

🔹 2. `Choreographer` schedules a vsync callback

```java
mChoreographer.postCallback(CALLBACK_TRAVERSAL, mTraversalRunnable, null);
```

- Choreographer is a **frame scheduler** tied to display vsync (typically 60fps or 120fps).
- It posts `TraversalRunnable` that ends up calling `performTraversals()`.

🔹 3. `ViewRootImpl.performTraversals()` runs layout, measure, draw

```java
view.measure(...) → view.layout(...) → view.draw(canvas)
```

- Calls all the `View` tree methods to measure and layout.
- Then calls `draw(Canvas)` which draws everything onto a `Canvas`.

🔹 4. How does `Canvas` draw to the screen?

There are **two paths**:

A. Software rendering:

- Uses **Skia** and renders to a `Bitmap` backed by the `Surface`.
- Used in safe mode or non-hardware accelerated Views.

B. Hardware acceleration (most apps):

- Uses **RenderThread** + **OpenGL ES** or **Vulkan**.

- Backed by **`HardwareRenderer`** and **`ThreadedRenderer`**.

```java
mAttachInfo.mThreadedRenderer.draw(...) → RenderThread → EGL → Surface
```

🔹 5. `RenderThread` prepares the frame (Hardware-accelerated apps)

- **`RenderThread`** runs in parallel to the main thread.
- Uses **OpenGL or Vulkan** to render Views into buffers attached to the Surface.
- It calls **EGL draw calls** into a framebuffer (double/triple buffer).

🔹 6. Final frame is posted to `SurfaceFlinger`

- **`eglSwapBuffers()`** posts the final frame to Surface.
- Now **SurfaceFlinger** gets the buffer from your Surface.

🌄 What SurfaceFlinger Does

SurfaceFlinger collects surfaces from:

- Apps
- System UI (status bar, navigation bar)
- Input layers
- Wallpaper, etc.

Then it:

1. **Composites all layers**
2. **Applies transformations** (rotation, scaling, etc.)
3. **Uses HWC (Hardware Composer HAL)** to push to display

📦 Example Timeline (Simplified at 60fps)

Time (ms)	Component	Description
0 ms	App	Calls **`requestLayout()`**
1 ms	ViewRootImpl	Schedules frame via **`Choreographer`**

Time (ms)	Component	Description
16 ms	VSync	**`Choreographer.doFrame()`** triggers traversal
17–22 ms	Layout & Draw	Views drawn to Surface (Canvas or GPU)
23 ms	EGL	**`eglSwapBuffers()`** posts buffer
24–32 ms	SurfaceFlinger	Composites and shows on screen

🖼️ Visualization

```css
App code
  ↓
requestLayout()
  ↓
Choreographer schedules VSync
  ↓
View hierarchy measured + laid out + drawn
  ↓
HardwareRenderer or Canvas draws to Surface
  ↓
SurfaceFlinger composites Surfaces
  ↓
Display shows final frame
```

📌 Key Components Recap

Component	Role
`ViewRootImpl`	Bridge between View system and Surface
`Choreographer`	Schedules frame on vsync
`RenderThread`	Offloads rendering from UI thread (OpenGL/Vulkan)
`SurfaceFlinger`	Composes final frame from all surfaces
`SurfaceControl`	Low-level wrapper to manage a layer
`Surface`	Actual drawing buffer
`EGL`	Interface to GPU for rendering
`HWC`	Talks to display hardware (via HAL)

➕ Want to Go Deeper?

I can explain any of these in detail:

- **`RenderThread`** architecture and EGL setup
- SurfaceFlinger composition pipeline

- How VSync and triple buffering reduce jank
- Difference between SurfaceView and TextureView
- GPU memory and buffer management (**`GraphicBuffer`**, **`BufferQueue`**)
- Hardware Composer HAL (**`composer@2.x`**)

Just tell me which one you want to dive into.

Surface display process summary:

After creating PhoneWindow in the App process, ViewRoot is created. The creation of ViewRoot creates a Surface, which is actually empty. When communicating with WindowManagerService, copyFrom() a NativeSurface is used to communicate with SurfaceFlinger.

About Native Window:

Native Window is a bridge between OpenGL and the local window system. The entire GUI system requires at least two types of local windows:

(1) Manager-oriented (SurfaceFlinger)

SurfaceFlinger is the manager of all UI interfaces in the system and needs to directly or indirectly hold a "local window". This local window is FramebufferNativeWindow.

(2) Application-oriented

This type of local window is Surface.

Normally, when generating an APK application according to the SDK wizard, third-party graphics libraries such as Skia are used. However, for application developers who want to use OpenGL ES to complete complex interface rendering, Android also provides encapsulated GLSurfaceView (or other methods) to achieve graphics display.

①FramebufferNativeWindow

EGL needs to use a native window to create an environment for OpenGL/OpenGL ES. Since OpenGL/ES supports multiple platforms, compatibility and portability are taken into consideration. The data types of local windows EGLNativeWindowType on different platforms are different.

The data type on the Android platform is ANativeWindow, which is like a "protocol" that specifies the form and function of a local window. ANativeWindow is the parent class of FramebufferNativeWindow.

In Android, due to multi-buffer technology, the buffers managed by

EGLNativeWindowType are at least 2 and at most 3.

FramebufferNativeWindow initialization requires Golloc support, and the steps are as follows:

Load the GRALLOC_HARDWARE_MODULE_ID module, see the previous section.
Open the fb and gralloc devices respectively. The opened devices are managed by the global variables fbDev and grDev.
Assign initial values to FramebufferNativeWindow according to the properties of the device.
Fill in the "protocol" in ANativeWindow according to the implementation of FramebufferNativeWindow
Some other necessary initialization

②The local window of the application - Surface

Surface also inherits ANativeWindow

```
class Surface : public ANativeObjectBase<ANativeWindow, Surface, RefBase>
```

Surface is the interface for all UI applications in the Android system, fulfilling the UI display requirements of the application process.

It provides a drawing board for upper-layer implementations (primarily the Java layer) to draw images. SurfaceFlinger collects image data drawn by all applications in the system and displays it centrally on the physical screen. The Surface, while fulfilling this role, is essentially managed by the SurfaceFlinger service, which involves many cross-process communication details.

③Surface Creation:

The Surface obtains buffers through mGraphicBufferProducer, which are recorded in the data in mSlots. The initialization process for this core component, mGraphicBufferProducer, is as follows:

ViewRootImpl holds a Java-level Surface object (mSurface).

ViewRootImpl initiates a relayout request to WindowManagerService, at which point mSurface is assigned a valid value. The resulting SurfaceControl is then copied to mSurface via the Surface.copyFrom() function.

Thus, the Surface is managed by the SurfaceControl, which is created by the SurfaceComposerClient. The anonymous Binder obtained by the SurfaceComposerClient is ISurfaceComposer, whose server-side implementation is SurfaceFlinger . The IGraphicBufferProducer object that the Surface relies on is implemented as a BufferQueue on the server side.

```
class SurfaceFlinger :
  public BinderService<SurfaceFlinger>, // Register as SurfaceFlinger in ServiceManager
  public BnSurfaceComposer, // The implemented interface is called ISurfaceComposer
```

④SurfaceFlinger Service Framework:

Buffer, Consumer, and Producer are the three participants in the "Producer-Consumer" model. How to coordinate their work is key to whether the application can display the UI properly.

> The buffer is BufferQueue, the Producer is the application, and the Consumer is SurfaceFlinger.

Great summary! Here's a **concise and clearer breakdown** of the entire **Surface display process** and **native window usage in Android**, tying everything together in a digestible way:

🔷 1. Surface Creation and Display Flow

💡 High-level steps:

1. **App starts Activity** → `Activity.makeVisible()`
2. `PhoneWindow` is created.
3. `ViewRootImpl` is created → owns a Java-level `Surface`.
4. `ViewRootImpl.relayoutWindow()` calls **WindowManagerService (WMS)**.
5. WMS requests a `SurfaceControl` from **SurfaceFlinger**.
6. SurfaceControl is wrapped into a **native** `Surface` via `Surface.copyFrom()`.
7. App renders into the `Surface` via Skia, Canvas, or OpenGL (via EGL).
8. **SurfaceFlinger** composites all surfaces and sends final frame to display.

🔷 2. What is a Layer?

- A **Layer** = `Surface` + `SurfaceControl`
- It represents **a unit of composition** that SurfaceFlinger can manage.
- Think of layers like transparent sheets stacked on top of each other — each app's surface is one layer.

🔷 3. Understanding Native Window Types

✅ `ANativeWindow`

- A **platform abstraction** used by EGL/OpenGL ES for rendering targets.
- Implemented as either `Surface` or `FramebufferNativeWindow`.

✅ Application-side: `Surface`

- Inherits from `ANativeWindow`
- Used by:

- Skia (Java Canvas)
- EGL/OpenGL via **`EGLWindowSurface`**
- MediaCodec output
- Behind it is a **`BufferQueue`**:
  - **Producer:** App (draws to buffer)
  - **Consumer:** SurfaceFlinger (composites it)

✅ System-side: `FramebufferNativeWindow`

- Also inherits **`ANativeWindow`**
- Used by SurfaceFlinger to directly render to the **physical framebuffer** (when HWComposer is not available).
- Usually involved in boot logo / recovery modes.

🔷 4. SurfaceFlinger Role

- SurfaceFlinger is the **central compositor** in Android.
- It:
  - Takes **all app surfaces** (each with a **`BufferQueue`**).
  - Reads their contents (consumer side).
  - Composites them using hardware composer (HWC) or GPU.
  - Displays final output on screen.

🔷 5. SurfaceControl & SurfaceComposerClient

- **`SurfaceComposerClient`** (in app or WMS) requests **`SurfaceControl`** from SurfaceFlinger.
- **`SurfaceControl`** creates/manages a **layer**.
- That control creates a **`BufferQueue`** and gives the producer end to the app (**`Surface`**).
- The consumer end is managed by SurfaceFlinger.

🔷 6. BufferQueue in Detail

- A **`BufferQueue`** is the **heart of the producer-consumer mechanism**.

Component	Role	Notes
App / Surface	Producer	Draws frames

Component	Role	Notes
SurfaceFlinger	Consumer	Reads frames for display
BufferQueue	Shared	Coordinates buffer handoff

- It maintains at least **2 or 3 buffers** (triple buffering).
- Uses **`gralloc`** to allocate buffers in shared memory (ashmem or dmabuf-backed).

🔷 7. EGL Integration

- EGL needs a **`native_window`** (i.e., **`ANativeWindow`**) to create a rendering surface:

  ```c
  EGLSurface eglSurface = eglCreateWindowSurface(eglDisplay, config, native_window, NULL);
  ```

- If you're using OpenGL ES in a game or GL-based app, you're rendering into a **`Surface`** (ANativeWindow) backed by a **`BufferQueue`**.

🔷 8. Simple Real-World Example

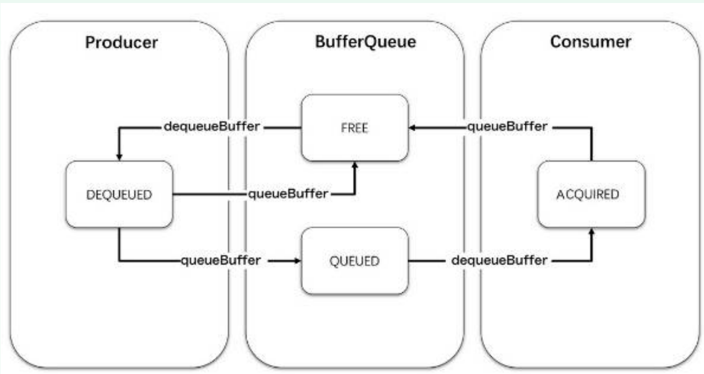
Let's say you open YouTube:

1. The **app creates a Window and Surface**.
2. Video rendering uses **MediaCodec** → **Surface**.
3. UI (play button, seek bar) is drawn on the same or another surface.
4. **SurfaceFlinger**:
   - Gets these surfaces.
   - Composites the video + UI layers.
   - Shows the result on screen.

🔚 Summary Diagram (simplified):

```makefile
App: ViewRootImpl
 └── Surface (ANativeWindow) ──> Producer ──┐
                                          │
SurfaceFlinger                            BufferQueue
                                          │
 └── Framebuffer / DisplayController <── Consumer
```

Let me know if you want this broken down visually with a diagram or want to dive into a specific part (e.g., **`SurfaceControl`**, **`gralloc`**, **`BufferQueue`**, etc.)



Surface provides a BufferQueue internally, which forms a producer-consumer model with the upper layer and SurfaceFlinger. The upper layer corresponds to the Producer, and SurfaceFlinger corresponds to the Consumer. The three are connected through the Buffer, and each Buffer has four states:

Free: can be used by upper layers;
Dequeued: out of the queue and being used by the upper layer;
Queued: Enqueued, upper layer drawing has been completed, waiting for SurfaceFlinger synthesis;
Acquired: Acquired, SurfaceFlinger is holding the Buffer for synthesis;
This cycle forms a process in which the Buffer is recycled (FREE->DEQUEUED->QUEUED->ACQUIRED->FREE).

The mSlots array in BufferQueue is used to manage the buffer in the period, and the maximum container is 32. The space of the data buffer is dynamically allocated. Both the application and SurfaceFlinger use OpenGL ES to complete the UI display. The Layer class represents a "layer" in SurfaceFlinger, which generally represents a "screen". The final display result on the physical screen is formed by processing and superimposing all the "screens" that exist in the system at the same time.

Excellent explanation — and the **diagram** you've shared clearly visualizes the **BufferQueue lifecycle** and its **producer-consumer** mechanics.

Let's walk through the **full cycle of buffer states** and how each component interacts:

🔄 Buffer Lifecycle in BufferQueue (via Surface)

📦 States:

1. **FREE**
   - The buffer is available for the producer (usually the app) to use.
   - Stored in a pool inside **`BufferQueue`**.
2. **DEQUEUED**
   - The app (producer) calls **`dequeueBuffer()`** to obtain a buffer.
   - The buffer is now *locked for writing*.
   - App performs rendering (Canvas, OpenGL, MediaCodec, etc.).
3. **QUEUED**
   - After drawing, the app calls **`queueBuffer()`** to submit it.
   - The buffer enters the *queue*, waiting to be processed by the consumer.
4. **ACQUIRED**
   - SurfaceFlinger (the consumer) calls **`dequeueBuffer()`** to obtain the ready buffer.
   - It performs compositing — blending it with other layers (via HWC/GPU).
5. **FREE** (again)
   - Once SurfaceFlinger is done rendering the frame, it calls **`queueBuffer()`** to return the buffer to the free pool.

🔗 Component Roles

Component	Role	Key Method Calls
Producer (App)	Renders UI	**`dequeueBuffer()`** → draw → **`queueBuffer()`**
BufferQueue	Mediator	Manages slot state transitions
Consumer (SurfaceFlinger)	Composites frames	**`dequeueBuffer()`** → render → **`queueBuffer()`**

🔢 Buffer Management: mSlots Array

- **`BufferQueue`** maintains **up to 32 slots** via **`mSlots[]`**.
- Each slot holds a reference to a buffer and its current **state**.
- Slots are reused to reduce buffer allocation overhead.
- Slots move through state transitions:
  `FREE → DEQUEUED → QUEUED → ACQUIRED → FREE`

🧠 Advanced Notes

- **Triple Buffering** is commonly used:
  - 1 buffer being displayed (ACQUIRED)
  - 1 buffer being drawn into (DEQUEUED)
  - 1 buffer waiting to be displayed (QUEUED)
- **Backpressure**:
  - If the producer is too fast, it may be blocked on `dequeueBuffer()` until a buffer becomes FREE again.
  - Ensures synchronization with display refresh rate (e.g. 60Hz).
- **SurfaceFlinger's Layer**:
  - Each surface/layer in the system is represented by a `Layer` class in SurfaceFlinger.
  - Composited layers form the final screen output.

🖼️ Simplified Visual Summary

```sql
[App]            [BufferQueue]           [SurfaceFlinger]
   |                    |                          |
   |--dequeueBuffer-->  |                          |
   |                    |---> DEQUEUED             |
   |          (App renders into buffer)            |
   |--queueBuffer---->  |                          |
   |                    |---> QUEUED               |
   |                    |          <--dequeue--    |
   |                    |<---- ACQUIRED ----------|
   |                    |                          |
   |<----queueBuffer---|                           |
   |                    |---> FREE                 |
```

Let me know if you'd like an annotated version of the diagram you uploaded with these state transitions labeled — or a more detailed visual of `mSlots[]` and buffer slot management.