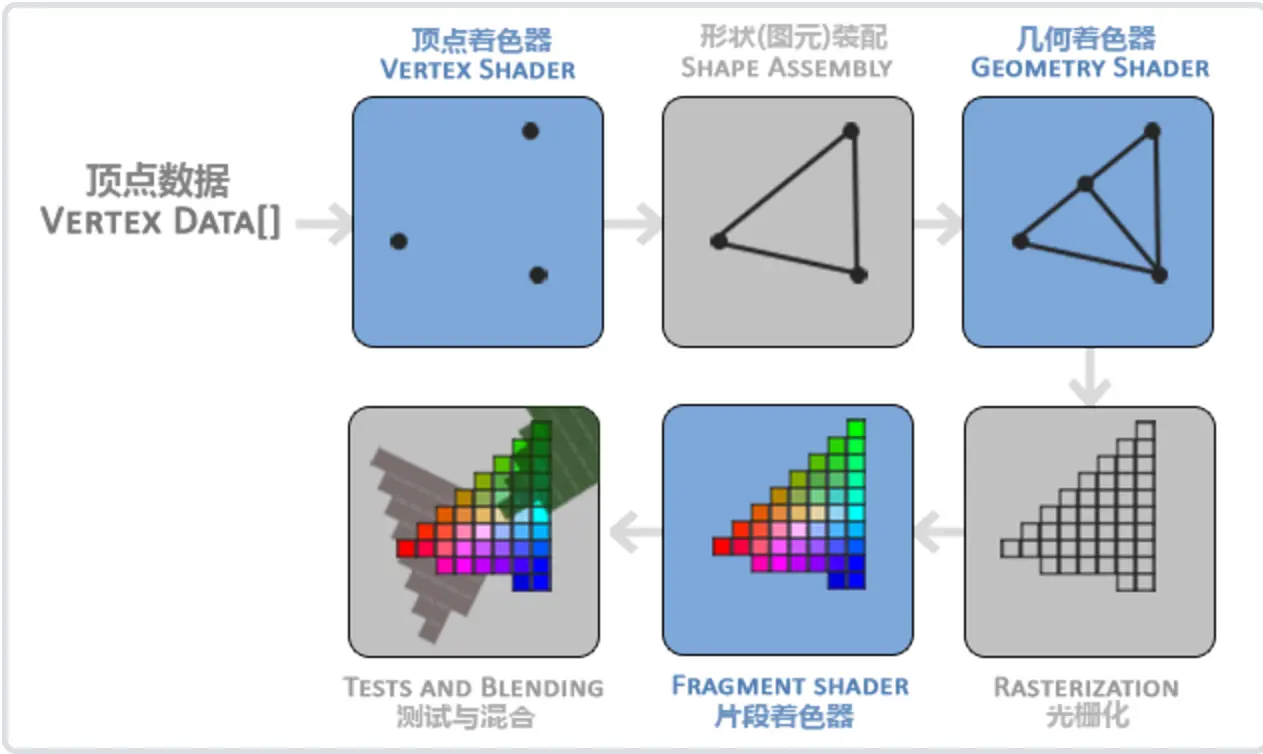
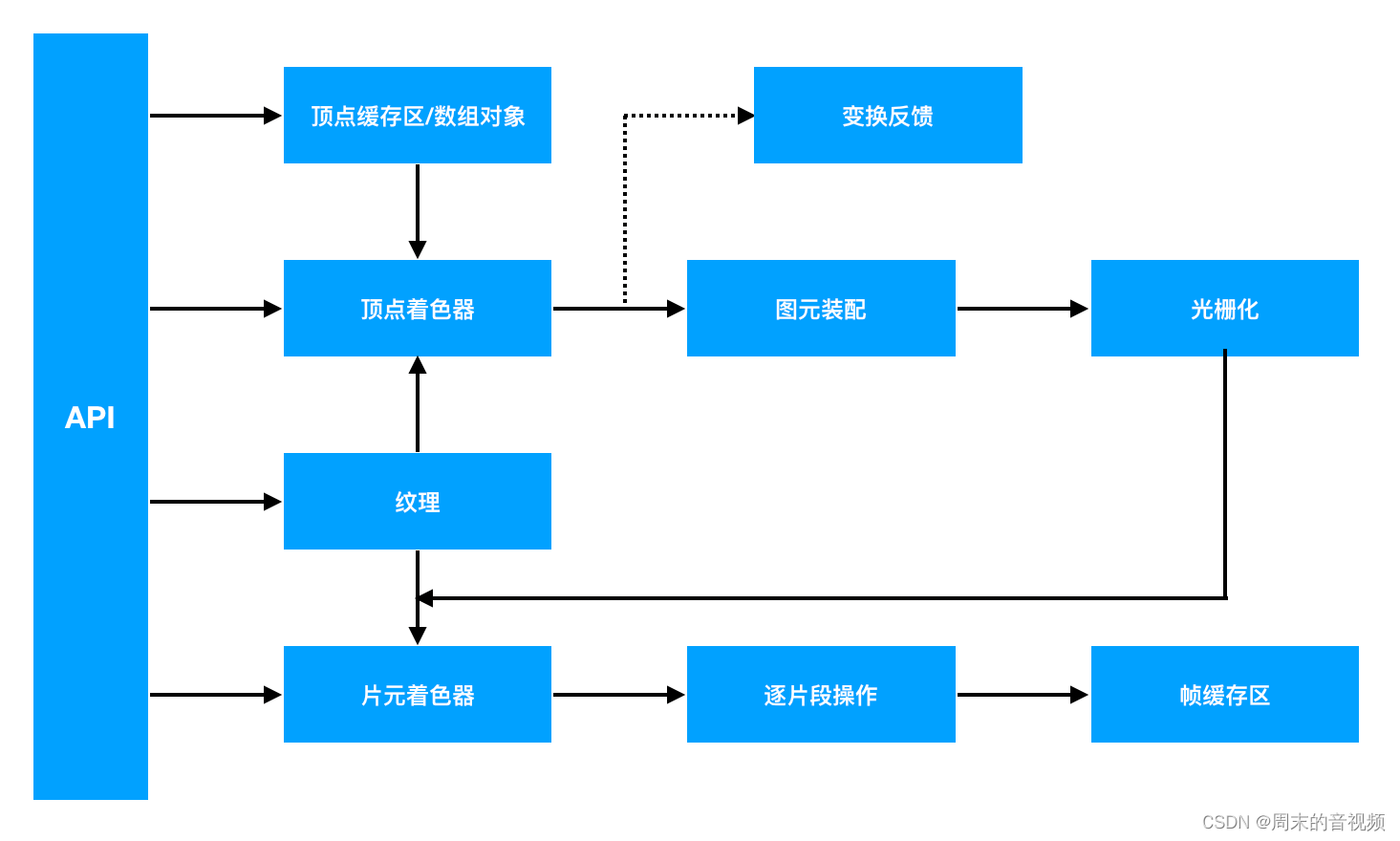
mipi相关的操作流程:

Andoid 显示系统



### **OpenGL ES 3.0图形管线**



1）、使用API创建顶点数据（纹理坐标也是顶点数据）存入顶点缓冲区/数组对象，顶点缓冲区是GPU显存上的区间，顶点数组存储在内存中，数据会传给顶点着色器；

（2）、顶点着色器对glPiont赋值、执行旋转缩放变换等操作，之后进行图元装配；

（3）、图元装配中设置顶点之间的连接方式来确定绘制方式，然后绘制图形，之后对图形进行光栅化；

（4）、光栅化是确定图形在屏幕上的像素点，便于后续片元着色器填充颜色；

（5）、纹理坐标数据是不断变化的，需要通过attribute属性通道交给顶点着色器往下传递到片元着色器；

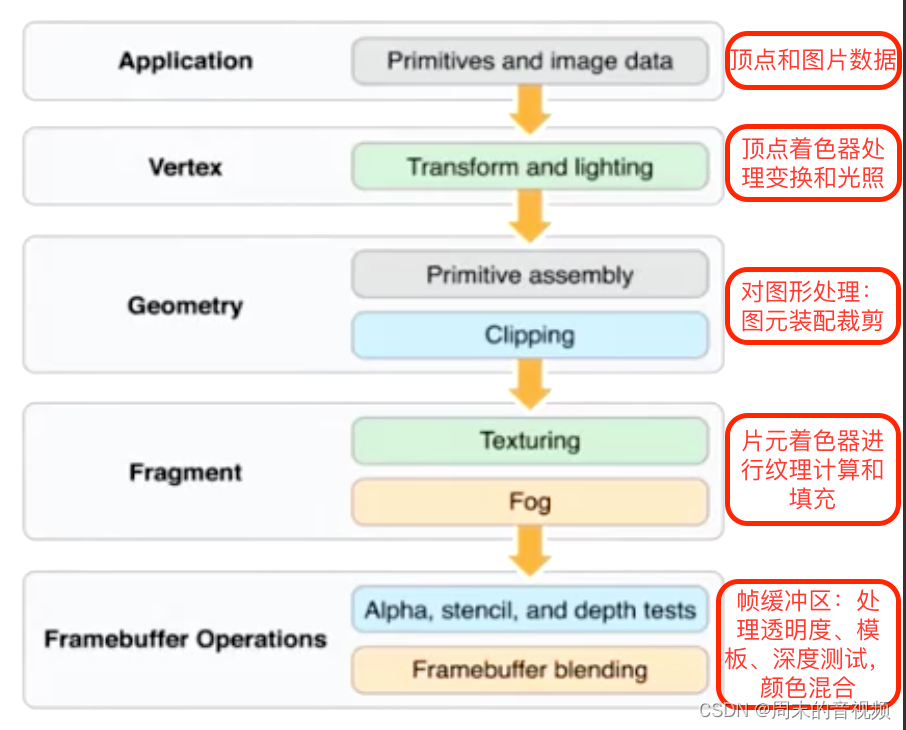
（6）、片元着色器对各个像素点填充颜色，可以对图形填充纯色，也可以从纹理数据中获取对应位置的颜色填充；

（7）、逐片元操作是将图形中的每一个像素点填充颜色，把结果放在帧缓冲区中；

（8）、从帧缓冲区将结果显示到屏幕上。tips：图元装配、光栅化、逐片元操作和帧缓冲区是不由开发者参与的，OpenGL会自动计算。

、

### **OpenGL ES图形管道**



（1）、application应用：程序代码将图片原始数据如顶点数据和图片数据传递给顶点着色器；

（2）、Vertex顶点着色器：顶点着色器处理旋转缩放等变换数据，需要光照的添加光照数据，将这些数据往下传递进行图元装配；

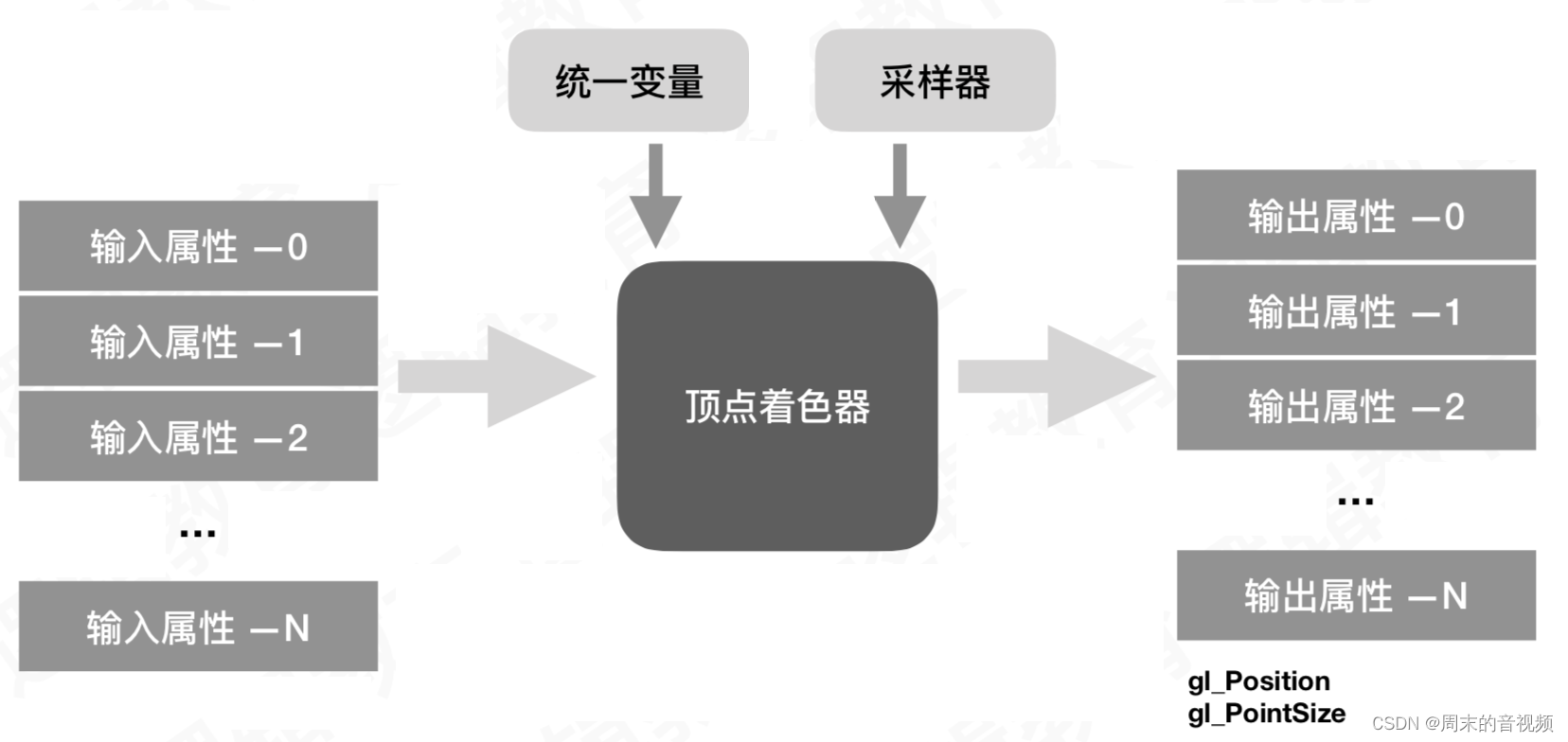
（3）、Geometry几何处理：对图形的处理，包括图元装配和裁剪，然后交给片元着色器处理；

（4）、Fragment片元着色器：片元着色器计算纹理坐标颜色并填充像素点，将结果存入帧缓冲区；

（5）、Framebuffer Operations帧缓冲区：处理图片透明度、模板、开启深度测试，最后进行颜色混合，从帧缓冲区读取数据并显示到屏幕。

## **OpenGL ES几个重要概念**

### **1、顶点着色器**



顶点着色器的几个要点：

（1）、着色器程序——输入顶点数据，描述顶点上执行操作的顶点着色器程序源代码/可执行文件；

（2）、顶点着色器输入[属性attribute]——用顶点数组提供每个顶点的数据，attribute不能修饰片元着色器里的变量；

（3）、统一变量[uniform]——顶点/片元着色器使用的不变数据，比如客户端代码中计算好的旋转矩阵，通过uniform传输到GLSL中让三角形的每一个顶点都应用旋转矩阵；

（4）、采样器——处理纹理时的一个特殊的变量，获取纹理时不会传递纹理文件纹理数据或者纹理指针到GLSL，而是使用纹理ID在内存中查找对应的纹理。代表顶点着色器使用纹理的特殊统一变量类型；顶点着色器输出属性：

顶点着色器实际就是获取到最终的结果给gl\_Position赋值，使得计算变换之后的顶点生效，也可以对gl\_PointSize赋值更改点的大小。

顶点着色器功能：

（1）、矩阵变换位置

（2）、计算光照公式生成逐顶点颜色

（3）、生成/变换纹理坐标总结: 它可以用于执行自定义计算,实施新的变换,照明或者传统的固定功能所不允许的基于顶点的效果。

attribute vec4 position; //attribute属性变量修饰符，vec4是数据类型表示四维向量，顶点坐标

attribute vec2 texCoordinate; //vec2二维向量，纹理坐标是二维纹理

uniform mat4 rotateMatrix; //uniform统一通道修饰符，mat4四行四列的矩阵，旋转矩阵

varying lowp vec2 varyTexCoord; //纹理坐标传递到片元着色器，通过varying通道传递，lowp表示低精度

void main() {

varyTexCoord = texCoordinate; //赋值后传递给片元着色器

vec4 vPos = position;

vPos = vPos \* rotateMatrix; //每一个顶点与旋转矩阵相乘得到新的顶点

gl\_Position = vPos; //赋值最终计算好的新顶点，gl\_Position是内建变量，不需要定义只需要赋值

}

****GPU并行计算顶点，三角形的3个顶点并行计算，上面的代码并行执行3次****

****2、图元装配****

****顶点着色器之后,下一个阶段就是图元装配，确定图形的形状。****

****三种图元(Primitive)：点,线,三角形.****

****图元装配：将顶点数据计算成⼀个个图元.在这个阶段会执⾏裁剪、透视分割和 Viewport变换操作，这个过程开发者无法干预。****

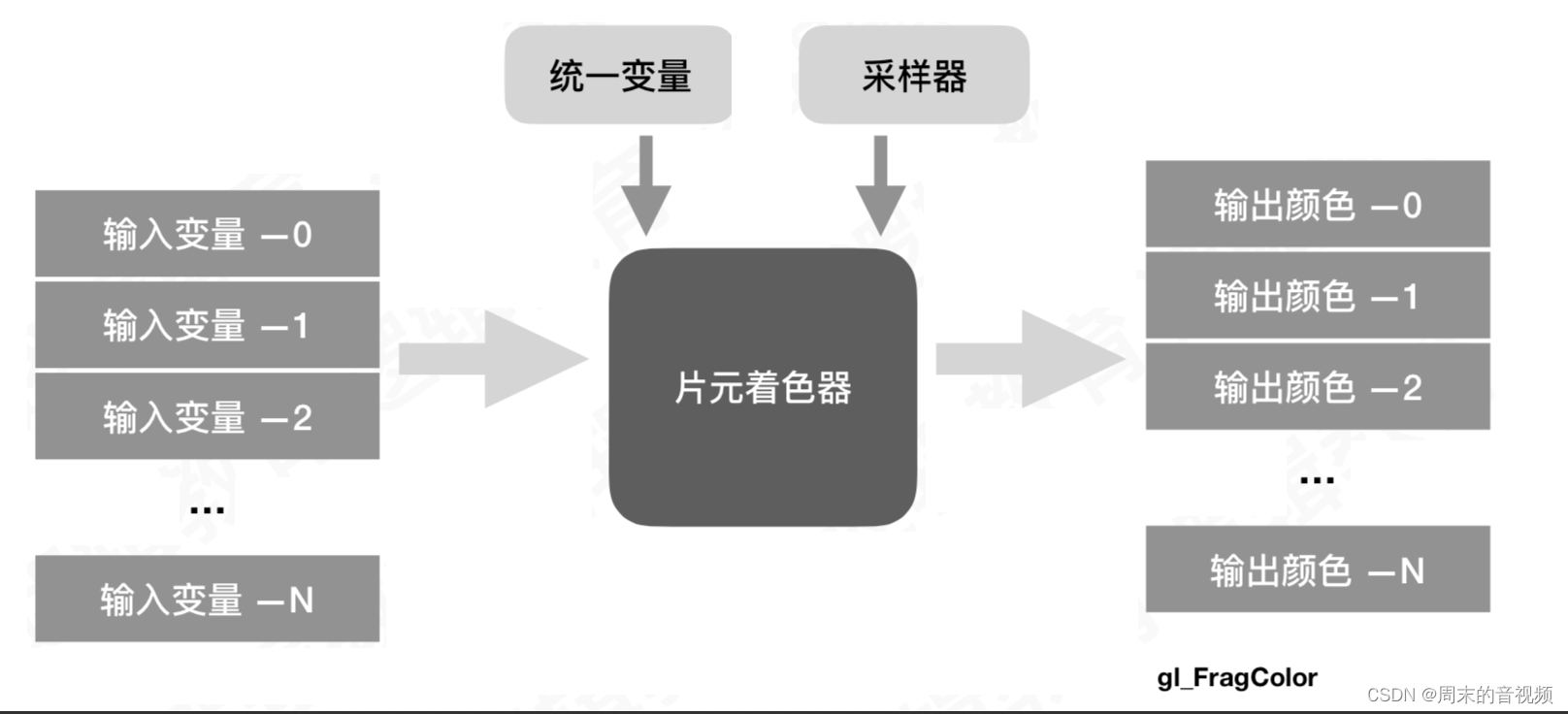
****图元类型和顶点确定将被渲染的单独图元。对于每个单独图元及其对应的顶点，图元装配阶段执行的操作包括：将顶点着色器的输出值执行裁剪、透视分割、视角变换后进入光栅化阶段。****

****3、光栅化****

****确定图形在屏幕上的像素点****

****在这个阶段绘制对应的图元(点/线/三⻆形)，光栅化就是将图元转化成一组二维片段的过程。而这些转化的片段将由片元着色处理。这些二维片段就是屏幕上可绘制的像素.****

### **4、片元着色器**



（1）、着色器程序——描述片段上执行操作的片元着色器程序源代码/可执行文件

（2）、输入变量——由顶点着色器传递进来 ，片元着色器没有属性attribute变量。光栅化单元用插值为每个片段生成的顶点着色器输出

（3）、统一变量(uniform)——顶点/片元着色器使用的不变数据，比如旋转矩阵

（4）、采样器——代表片元着色器使用纹理的特殊统一变量类型。获取到纹理ID从内存中查找对应的纹理片元着色器输出颜色：

片元着色器是输出颜色，通过计算颜色混合等最终输出一个像素点的颜色赋值给gl\_FragColor，让对应的这一个像素点得到一种颜色。

片元着色器功能：

（1）、计算颜色

（2）、获取纹理值，从图片中获取对应像素点的颜色值

（3）、往像素点中填充颜色值(纹理值/颜色值);

总结: 它可以用于图片/视频/图形中每个像素的颜色填充(比如给视频添加滤镜,实际上就是将视频中每个图片的像素点颜色填充进行修改.)

varying lowp vec2 varyTexCoord; //从顶点着色器传过来的纹理坐标，需要与顶点着色器的定义完全一致

uniform sampler2D colorMap; //由客户端通过uniform传递过来的纹理采样器，拿到对应的纹理颜色值，取得纹素

void main() {

//texture2D(纹理采样器, 纹理坐标);获取对应位置对应坐标上的颜色值，取得纹素

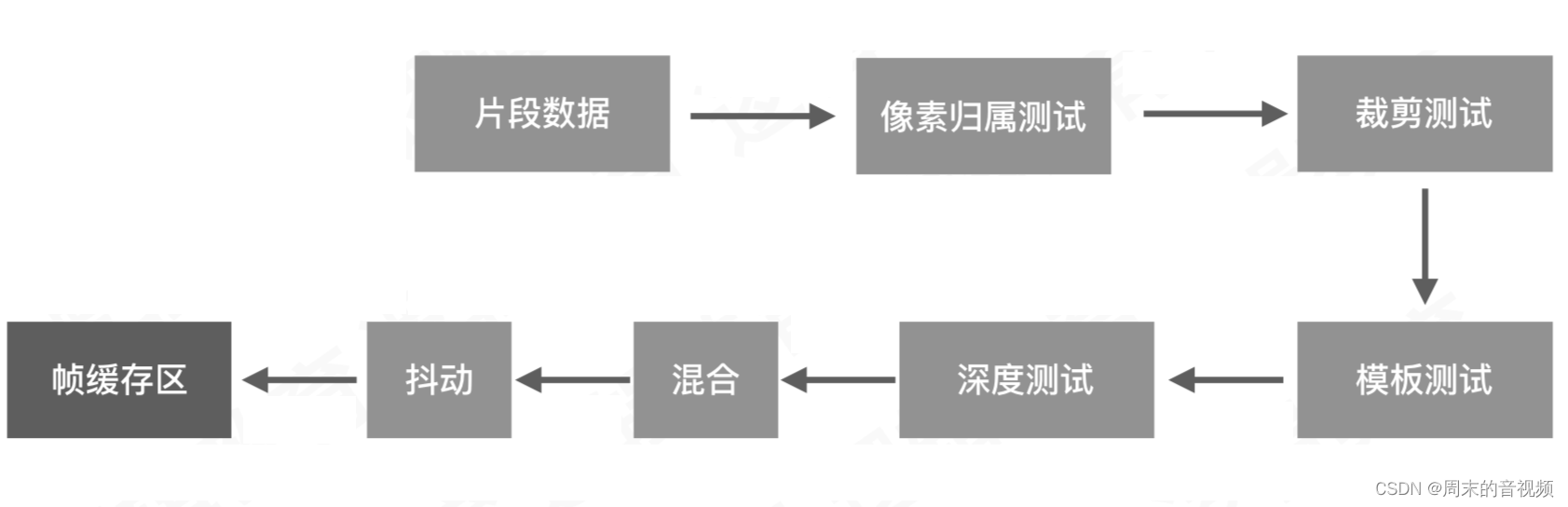
gl\_FragColor = texture2D(colorMap, varyTexCoord); //gl\_FragColor内建变量赋值，类型是vec4四维向量

}

****片元着色器执行次数跟图片像素点的个数一样，并行执行每一个像素点获取颜色，让每一个像素点得到颜色值。****

### ****5、逐片段操作****

        片元着色器取出颜色后，进行逐片段操作



拿到片段数据之后执行像素归属测试

（1）、像素归属测试：确定帧缓存区中位置(Xw,Yw)的像素目前是不是归属于OpenGL ES所有。 例如，如果一个显示OpenGL ES帧缓存区View被另外一个View 所遮蔽。则窗口系统可以确定被遮蔽的像素不属于OpenGL ES上下文，从而不全显示这些像素。像素归属测试是OpenGL ES 的一部分，它不由开发者人为控制，而是由OpenGL ES 内部进行。

（2）、裁剪测试：裁剪测试确定(Xw,Yw)是否位于作为OpenGL ES状态的一部分裁剪矩形范围内。如果该片段位于裁剪区域之外，则被抛弃。

（3）、深度测试：输入片段的深度值进一步比较，确定片段是否拒绝测试

（4）、混合：混合将新生成的片段颜色与保存在帧缓存的位置的颜色值组合起来。与片元着色器的颜色混合不同

（5）、抖动：抖动可用于最小化因为使用有限精度在帧缓存区中保存颜色值而产生的伪像。

6、OpenGL 上下文

OpenGL 是一个仅仅关注图像渲染的图像接口库，在渲染过程中它需要将顶点信息、纹理信息、编译好的着色器等渲染状态信息存储起来，而存储这些信息的数据结构就可以看作 OpenGL 的上下文。

调用任何 OpenGL 函数前，必须已经创建了 OpenGL Context，GL Context 存储了OpenGL 的状态变量以及其他渲染有关的信息。OpenGL 是个状态机，有很多状态变量，是个标准的过程式操作过程，改变状态会影响后续所有操作，这和面向对象的解耦原则不符，毕竟渲染本身就是个复杂的过程。

OpenGL 采用 Client-Server 模型来解释 OpenGL 程序，即 Server 存储 GL Context（可能不止一个），Client 提出渲染请求，Server 给予响应，一般 Server 和 Client 都在我们的 PC 上，但 Server 和 Client 也可以是通过网络连接。

之后的渲染工作就要依赖这些渲染状态信息来完成，当一个上下文被销毁时，它所对应的 OpenGL 渲染工作也将结束。

7、EGL(Embedded Graphics Library)

在 OpenGL 的设计中，OpenGL 是不负责管理窗口的，窗口的管理交由各个设备自己来完成，具体来讲，IOS 平台上使用 EAGL 提供本地平台对 OpenGL 的实现，在 Android 平台上使用 EGL 提供本地平台对 OpenGL 的实现。EGL 是 OpenGL ES 和 Android 底层平台视窗系统之间的接口，在 OpenGL 的输出与设备屏幕之间架接起一个桥梁，承担了为 OpenGL 提供上下文环境以及管理窗口的职责。

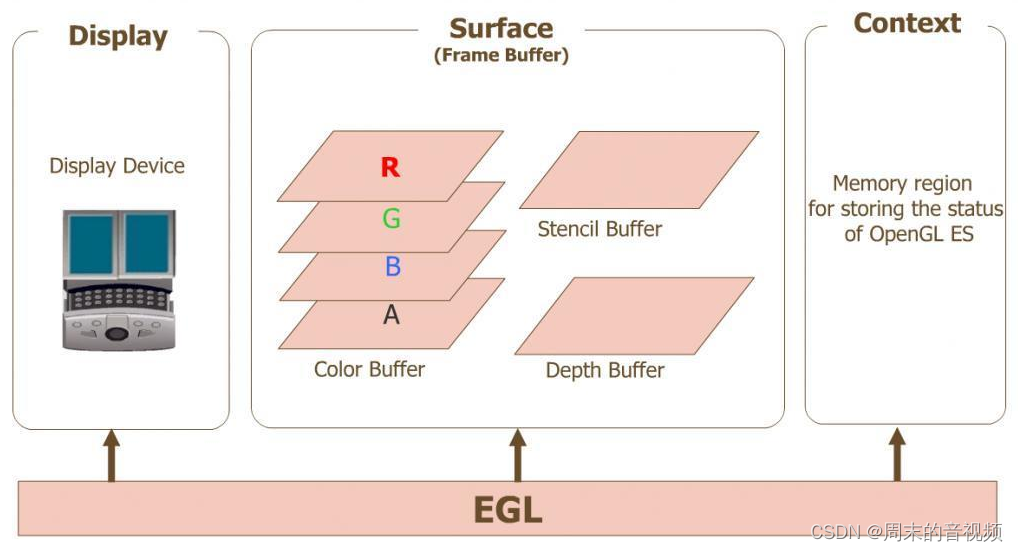
EGL 为双缓冲工作模式，即有一个 Back Frame Buffer 和一个 Front Frame Buffer，正常绘制的目标都是 Back Frame Buffer，绘制完成后再调用 eglSwapBuffer API，将绘制完毕的 FrameBuffer 交换到 Front Frame Buffer 并显示出来。

从代码层面来看，OpenGL ES 的 opengles 包下定义了平台无关的绘图指令，EGL（javax.microedition.khronos.egl）则定义了控制 displays，contexts 以及 surfaces 的统一的平台接口。

（1）、Display（EGLDisplay） 是对实际显示设备的抽象

（2）、Surface（EGLSurface）是对用来存储图像的内存区域 FrameBuffer 的抽象，包括 Color Buffer、Stencil Buffer、Depth Buffer

（3）、Context（EGLContext）存储 OpenGL ES 绘图的一些状态信息



framebufferNativeWindow

Egl 通过本地窗口为opengl es 创造环境

FramebufferNativeWindow.cpp F:\codec-pj\source\lc1860-4.4\frameworks\native\libs\ui 11487 2023/12/7 230

FramebufferNativeWindow 构造函数

FramebufferNativeWindow::FramebufferNativeWindow()

: BASE(), fbDev(0), grDev(0), mUpdateOnDemand(false)

--->hw\_get\_module(GRALLOC\_HARDWARE\_MODULE\_ID, &module)

--->framebuffer\_open(module, &fbDev)

--->gralloc\_open(module, &grDev);

--->mNumBuffers = fbDev->numFramebuffers;

--->buffers[i] = new NativeBuffer(

fbDev->width, fbDev->height, fbDev->format, GRALLOC\_USAGE\_HW\_FB);

--->grDev->alloc(grDev,

fbDev->width, fbDev->height, fbDev->format,

GRALLOC\_USAGE\_HW\_FB, &buffers[i]->handle, &buffers[i]->stride);

--->ANativeWindow::setSwapInterval = setSwapInterval;

--->ANativeWindow::dequeueBuffer = dequeueBuffer;

--->ANativeWindow::queueBuffer = queueBuffer;

--->ANativeWindow::query = query;

--->ANativeWindow::perform = perform;

--->ANativeWindow::dequeueBuffer\_DEPRECATED = dequeueBuffer\_DEPRECATED;

--->ANativeWindow::lockBuffer\_DEPRECATED = lockBuffer\_DEPRECATED;

--->ANativeWindow::queueBuffer\_DEPRECATED = queueBuffer\_DEPRECATED;

Opengl es通过dequeueBuffer分配一个用于渲染的缓冲区

FramebufferNativeWindow.cpp F:\codec-pj\source\lc1860-4.4\frameworks\native\libs\ui 11487 2023/12/7 230

FramebufferNativeWindow::dequeueBuffer(ANativeWindow\* window,

ANativeWindowBuffer\*\* buffer, int\* fenceFd)

--->FramebufferNativeWindow\* self = getSelf(window);

--->self->mCondition.wait(self->mutex);

--->\*buffer = self->buffers[index].get();

buffer显示到display设备

FramebufferNativeWindow.cpp F:\codec-pj\source\lc1860-4.4\frameworks\native\libs\ui 11487 2023/12/7 230

FramebufferNativeWindow::queueBuffer(ANativeWindow\* window,

ANativeWindowBuffer\* buffer, int fenceFd)

--->framebuffer\_device\_t\* fb = self->fbDev;

---> res = fb->post(fb, handle);//调用hal层的接口实现

framebuffer\_device.cpp F:\codec-pj\source\lc1860-4.4\hardware\leadcore\libgralloc 19234 2024/9/10 512

--->m->base.unlock(&m->base, m->currentBuffer);

vsync模式

--->gralloc\_mali\_vsync\_report(MALI\_VSYNC\_EVENT\_BEGIN\_WAIT);

--->ioctl(m->framebuffer->fd, COMIPFB\_IOCTL\_SET\_FPS, HIGH\_FPS)

--->ioctl(m->framebuffer->fd, COMIPFB\_IOCTL\_GET\_FPS, &fps);

--->ioctl(m->framebuffer->fd, FBIOPUT\_VSCREENINFO, &m->info)

--->gralloc\_mali\_vsync\_report(MALI\_VSYNC\_EVENT\_END\_WAIT);

--->m->currentBuffer = buffer;

非vsync模式

--->memcpy(fb\_vaddr, buffer\_vaddr, m->finfo.line\_length \* m->info.yres);

--->m->base.unlock(&m->base, buffer);

--->m->base.unlock(&m->base, m->framebuffer);

应用程序端的本地窗口 surface

Surface.cpp F:\codec-pj\source\lc1860-4.4\frameworks\native\libs\gui 25405 2023/12/20 664

1：集成ANativeWindow 所有需要实现ANativeWindow的协议

该协议包括：

window.h F:\codec-pj\source\lc1860-4.4\system\core\include\system 32710 2023/12/6 79

struct ANativeWindow

{

setSwapInterval :设置交换时间间隔

dequeueBuffer: EGL通过这个接口来申请一个buf

queueBuffer: 当EGL对渲染的buf完成后，调用这个接口来unlock和post buffer到显示设备

cancelBuffer: 取消一个已经dequeueBuffer的buf.

Perform: 执行本地窗口支持的各种操作。

Query: 查询本地窗口的信息。

}

Surface::Surface(

const sp<IGraphicBufferProducer>& bufferProducer,

bool controlledByApp)

: mGraphicBufferProducer(bufferProducer)

{

ANativeWindow::setSwapInterval = hook\_setSwapInterval;

ANativeWindow::dequeueBuffer = hook\_dequeueBuffer;

--->c->dequeueBuffer(buffer, fenceFd);

ANativeWindow::cancelBuffer = hook\_cancelBuffer;

ANativeWindow::queueBuffer = hook\_queueBuffer;

--->c->queueBuffer(buffer, fenceFd);

ANativeWindow::query = hook\_query;

ANativeWindow::perform = hook\_perform;

}

Surface成员

1：sp<IGraphicBufferProducer> mGraphicBufferProducer;

getIGraphicBufferProducer surface的核心，实现具体的协议。

IGraphicBufferProducer.h F:\codec-pj\source\lc1860-4.4\frameworks\native\include\gui 9587 2023/12/7 23

2：BufferSlot mSlots[NUM\_BUFFER\_SLOTS]; 存储buff

Surface::dequeueBuffer

--->result = mGraphicBufferProducer->dequeueBuffer(&buf, &fence, mSwapIntervalZero,

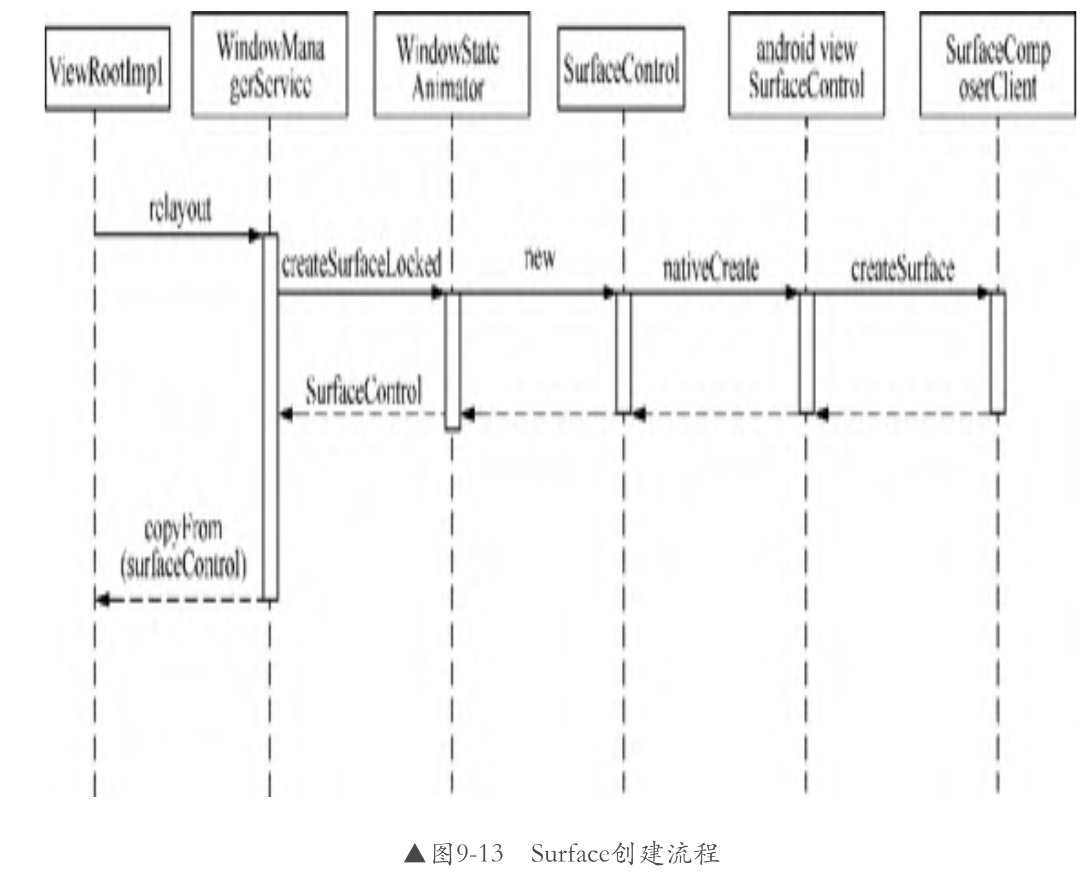
reqW, reqH, mReqFormat, mReqUsage);

--->sp<GraphicBuffer>& gbuf(mSlots[buf].buffer);

--->result = mGraphicBufferProducer->requestBuffer(buf, &gbuf);

--->\*buffer = gbuf.get();

应用层surface创建流程



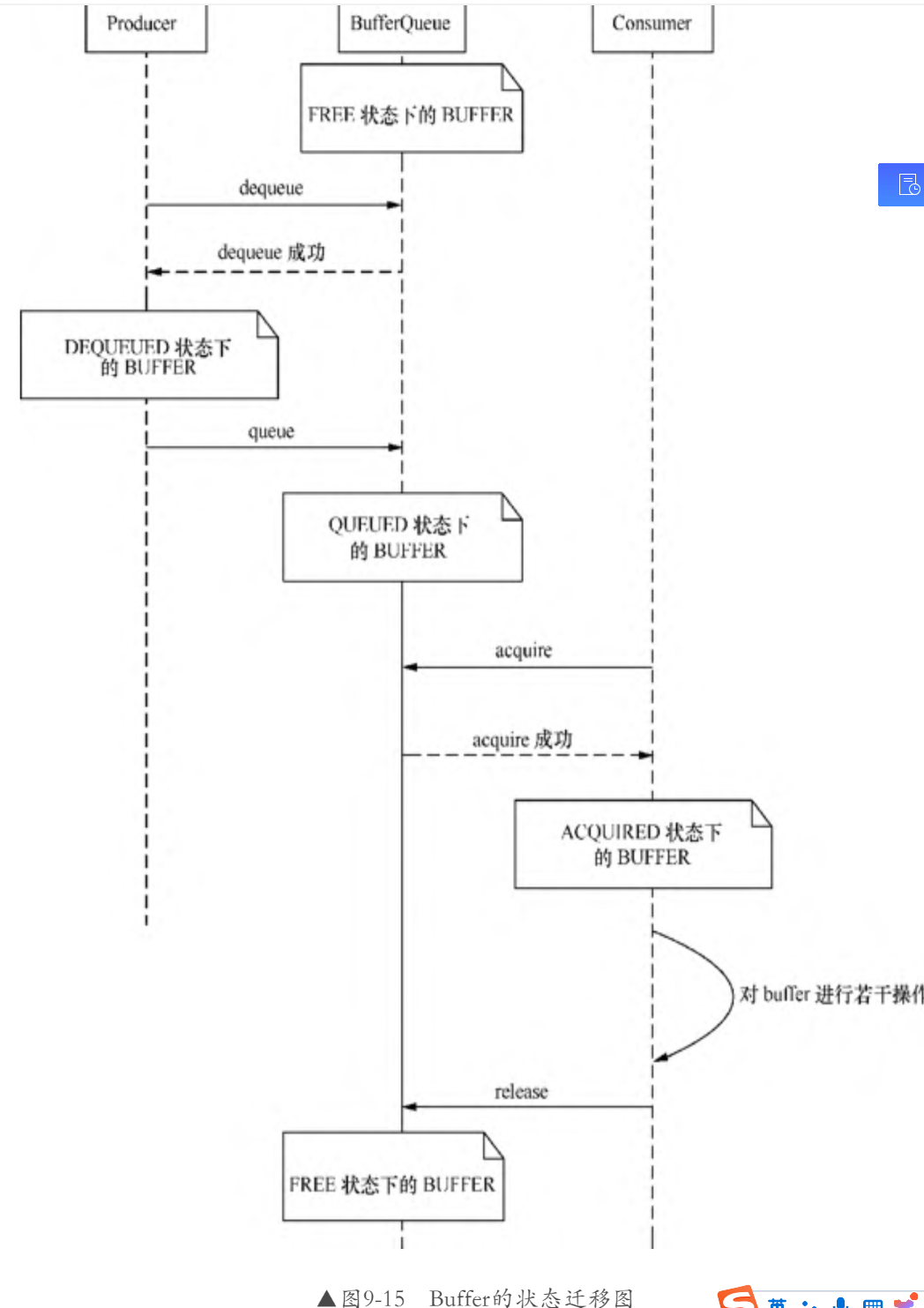
SurfaceComposerClient.cpp F:\codec-pj\source\lc1860-4.4\frameworks\native\libs\gui 22711 2023/12/7 447

sp<SurfaceControl> SurfaceComposerClient::createSurface

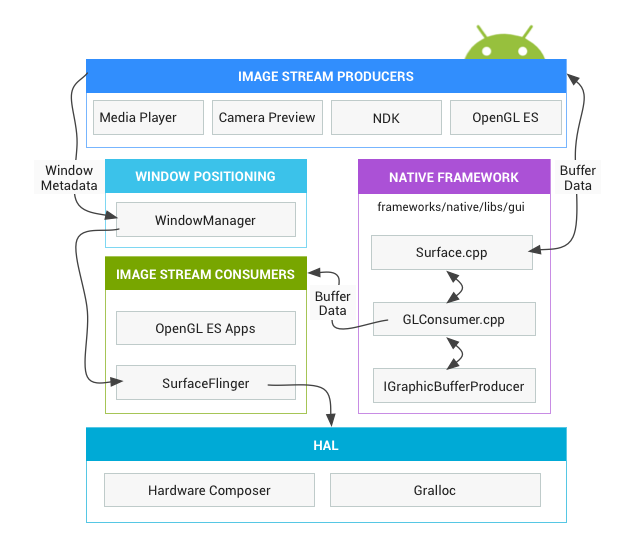
--->err = mClient->createSurface(name, w, h, format, flags,

&handle, &gbp);

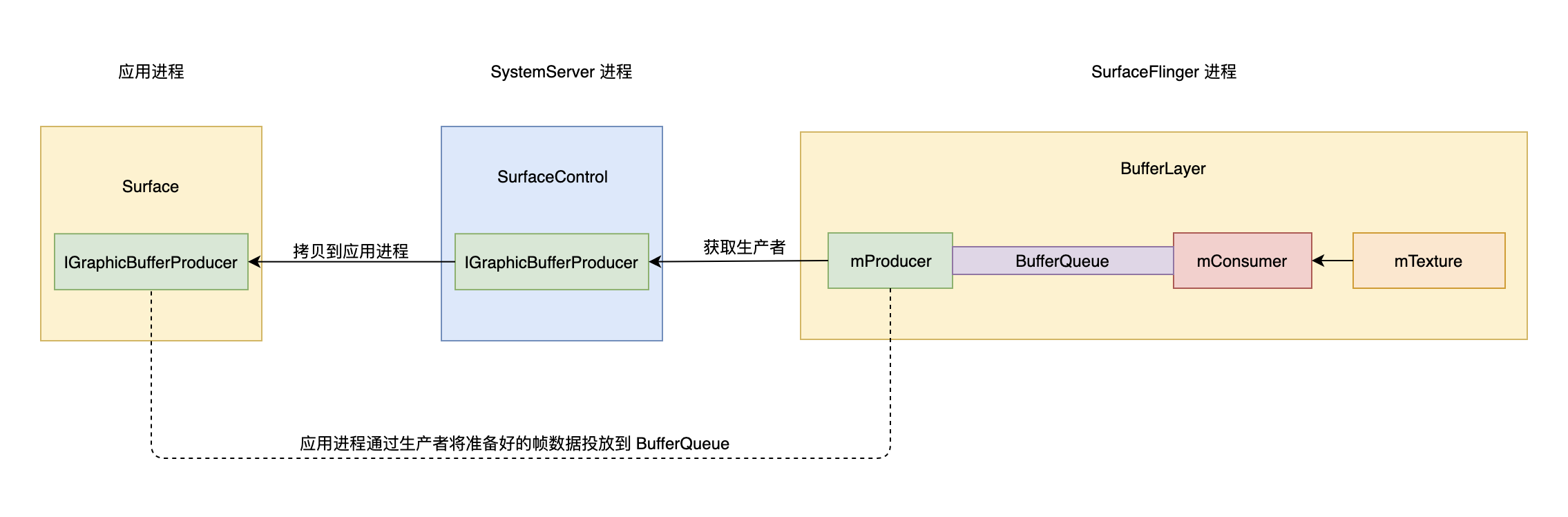
--->sur = new SurfaceControl(this, handle, gbp); //SurfaceControl本地对象



Android 图形渲染的组件结构图如下



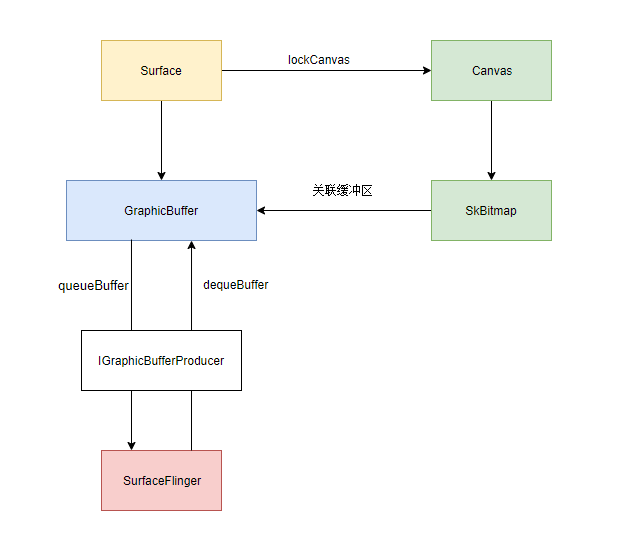
##### **Surface 的创建**



###### **软件渲染**

Android 端软件渲染的引擎为 Skia, 其****软件渲染的工作机制即使用 Canvas 将数据绘制到 Surface 的 GraphicBuffer 中****, 它的工作流程如下

* ****//////////////////////// Step1 /////////////////////////****
* Surface 锁定一个 GraphicBuffer 缓冲区
* 让 SkiaCanvas 的 SkBitmap 绑定缓冲区的共享内存
  + 意味着 Bitmap 的数据直接存储在缓冲区上了
* ****//////////////////////// Step2 /////////////////////////****
* 分发 View 绘制
* ****//////////////////////// Step3 /////////////////////////****
* Surface 释放 GraphicBuffer 缓冲区, 并将其推入 SurfaceFlinger 的渲染队列
* SurfaceFlinger 获取渲染数据输出到屏幕



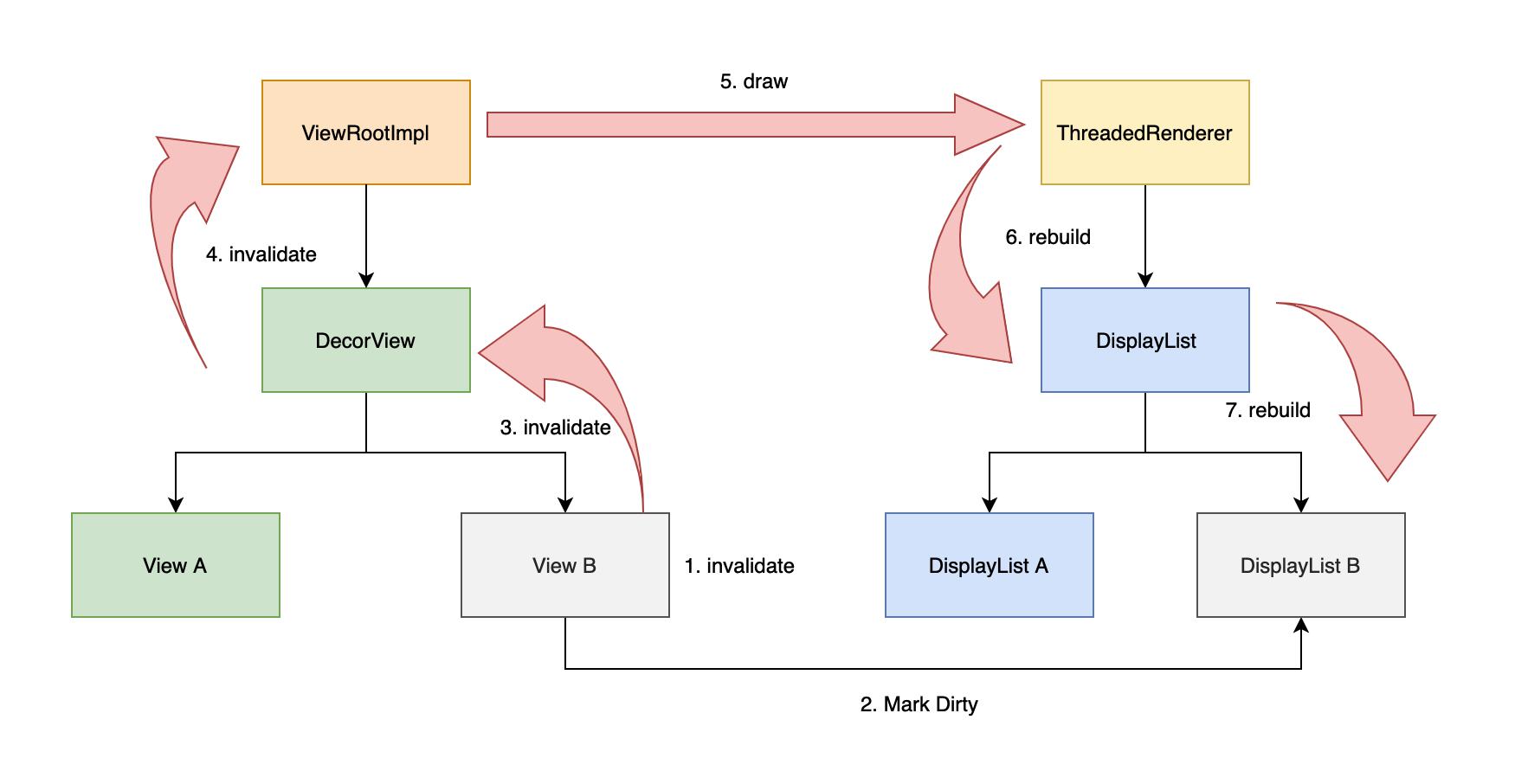
###### **硬件渲染**

硬件渲染, 即通过 GPU 来进行图形运算, 进行帧准备的操作

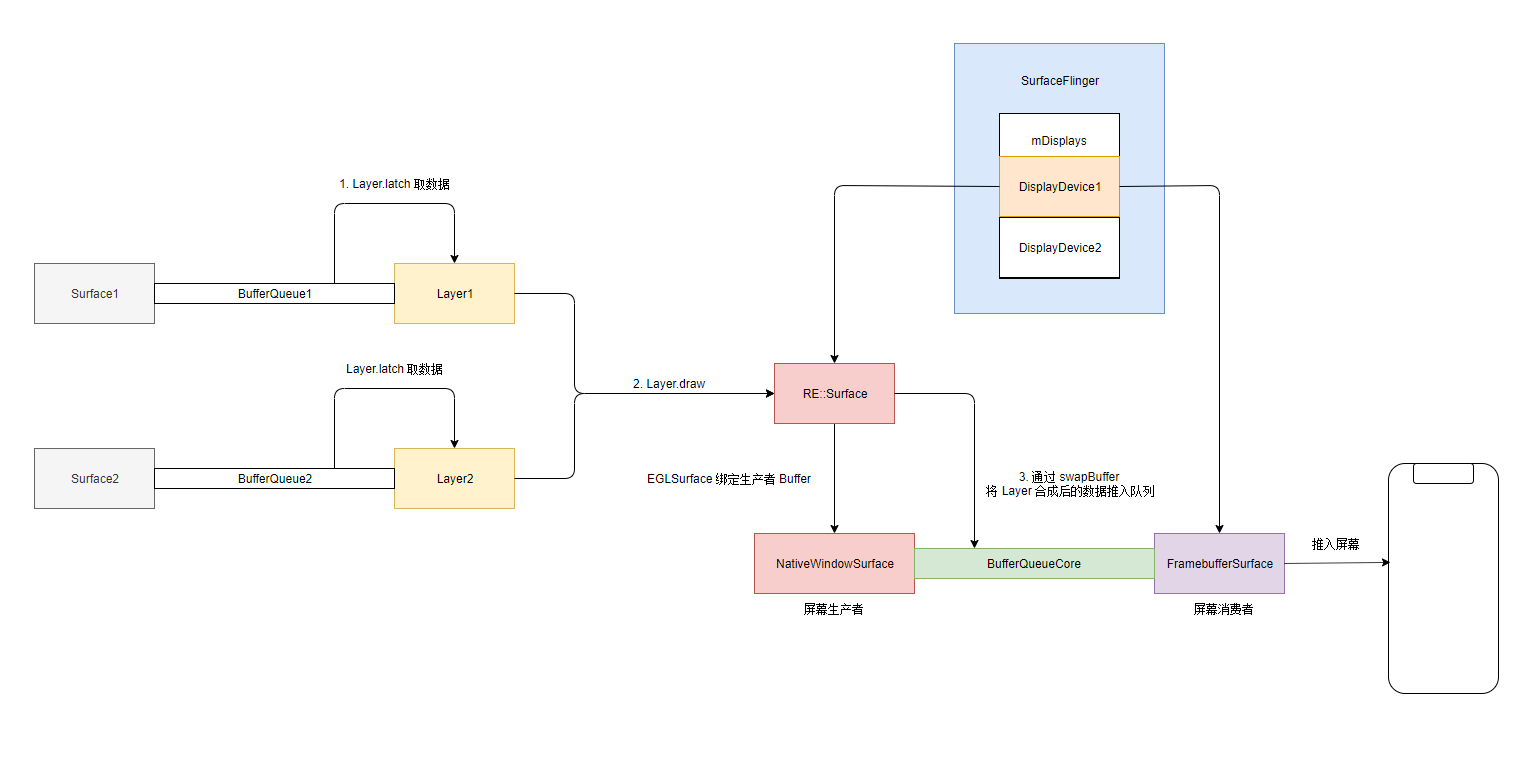
* 硬件渲染是由 ThreadRenderer 执行的, 它初始化的时候
  + 会将 Surface 绑定到 RenderPipeline 中
* 每个 View 中都有一个 RenderNode, 需要重绘时会将标记为 dirty 的 View 重构其 RenderNode 中的 DisplayList
  + DisplayList 用于捕获 View 的绘制动作, 并为真正开始绘制
* 构建完成之后, ThreadRenderer 会到 native 层的 RendererThread 中使用 OpenGL/Vulkan 执行 DisplayList 中的渲染操作
* 完成之后同过 swapBuffer 推到 Surface 对应 Layer 的缓冲队列

###### **两者差异**

* 从渲染机制
  + 硬件绘制使用的是 OpenGL/ Vulkan, 支持 3D 高性能图形绘制
  + 软件绘制使用的是 Skia, 仅支持 2D 图形绘制
* 渲染效率上: 硬件绘制较之软件绘制会更加流畅
  + 硬件绘制
    - 在 Android 5.0 之后引入了 RendererThread, 它将 OpenGL 图形栅格化的操作全部投递到了这个线程
    - 硬件绘制会跳过渲染数据无变更的 View, 直接分发给子视图
  + 软件绘制
    - 在将数据投入 SurfaceFlinger 之前, 所有的操作均在主线程执行
    - 不会跳过无变化的 View
* 从内存消耗上
  + 硬件绘制消耗的内存要高于软件绘制, 但在当下大内存手机时代, 用空间去换时间还是非常值得的
* 从兼容性上
  + 硬件绘制的 OpenGL 在各个 Android 版本的支持上会有一些不同, 常有因为兼容性出现的系统 bug
  + 软件绘制的 Skia 库从 Android 1.0 便屹立不倒, 因此它的兼容性要好于硬件绘制



### **消费进程**



通过 View 的 Traversals 操作, 我们应用进程准备的帧数据就投递到 Surface 对应在 SurfaceFlinger 的 Layer 中了, 当 SurfaceFlinger 的 EventThread-sf 发送一个 sf-VSYCN 信号时, 便会触发图层的合成, 将图层合成到屏幕的 Buffer 上

#### **1. 屏幕的生产者消费者**

* ****生产者为 NativeWindowSurface****
  + RE::Surface 它是一个 EGL 的工具类
    - ****其内部的 EGLSurface 绑定了生产者 NativeWindowSurface 中的缓冲区****
    - 通过 EGL 的渲染之后的图像数据会存储到 EGLSurface 绑定的缓冲区中
    - 通过 eglSwapBuffer, 将生产好的数据推送到屏幕的 Buffer 队列
* ****消费者为 FramebufferSurface****
  + 它负责从屏幕 Buffer 队列中取数据, 将其推送给硬件屏幕呈现出来

#### **2. 图层的合成**

sf-Vsync 的信号分发到 EventThread-sf从而 唤醒 SurfaceFlinger 的主线程, 处理 INVALIDATE 消息

* INVALIDATE 消息类型
  + ****从 Layer 的队列中锁定一个 GraphicBuffer****
* REFESH 消息类型
  + rebuildLayerStacks 负责 Layzer 的排序
    - ****每个显示设备的 Layer 按照 Z 轴进行排序****
  + doComposition 负责合并 Layer, 并且输出到屏幕
    - ****通过 RE::Surface 将 Layer 中的 Buffer 当做纹理合成到 EGLSurface 中****
    - ****调用 RE::Surface 的 swapBuffers, 将 EGLSurface 中的数据通过 IGraphicBufferProducer 推入屏幕渲染队列****
    - ****调用了 FramebufferSurface 的 advanceFrame, 将缓冲推给 HAL 进行展示****