The Android graphics path

in depth

License

About Chris Simmonds

- Consultant and trainer
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at http://2net.co.uk/

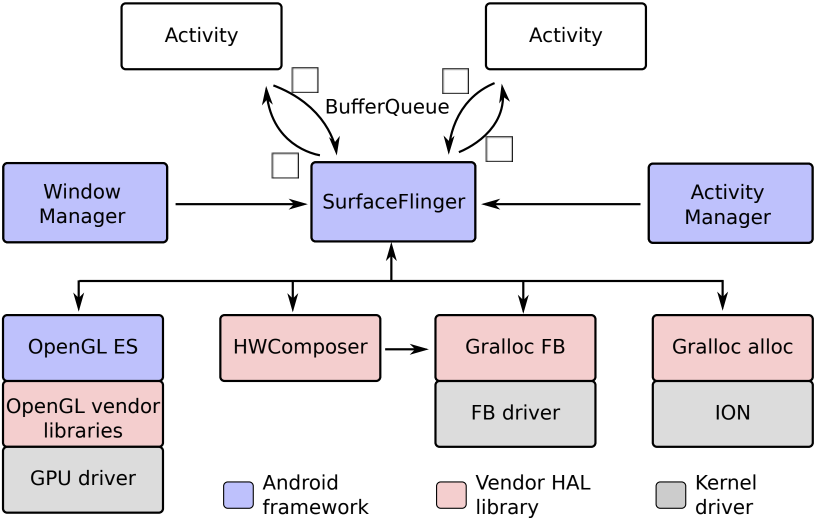
https://uk.linkedin.com/in/chrisdsimmonds/

https://google.com/+chrissimmonds

Overview

- The Android graphics stack changed a lot in Jelly Bean as a result of *project Butter*
- This presentation describes the current (JB) graphics stack from top to bottom
- Main topics covered
  - The application layer
  - SurfaceFlinger, interfaces and buffer queues
  - The hardware modules HWComposer and Gralloc
  - OpenGL ES and EGL

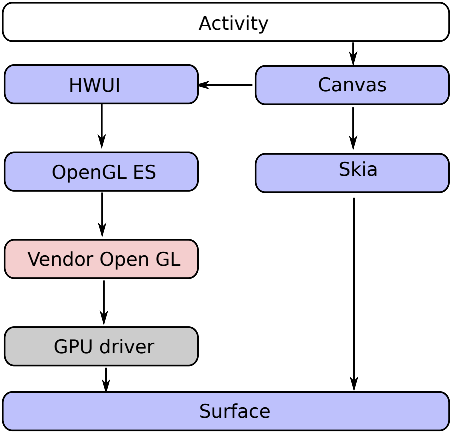
The big picture



Inception of a pixel

- Everything begins when an activity draws to a surface
- 2D applications can use
  - drawing functions in Canvas to write to a Bitmap: `android.graphics.Canvas.drawRect()`, `drawText()`, etc
  - descendants of the View class to draw objects such as buttons and lists
  - a custom View class to implement your own appearance and behaviour
- In all cases the drawing is rendered to a *Surface* which contains a *GraphicBuffer*

2D rendering path



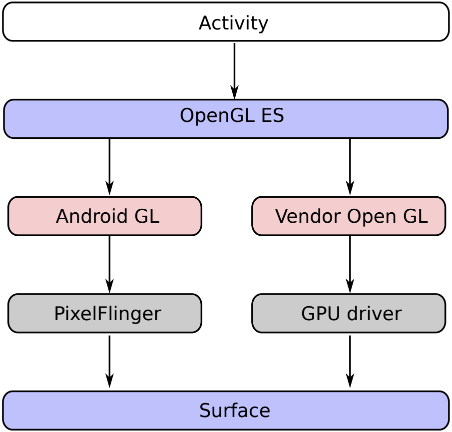
Skia and hwui

- For 2D drawing there are two rendering paths
  - hwui: (libwhui.so) hardware accelerated using OpenGL ES 2.0
  - skia: (libskia.so) software render engine
- hwui is the default
- Hardware rendering can be disabled per view, window, activity, application or for the whole device
  - Maybe for comparability reasons: hwui produces results different to skia in some (rare) cases

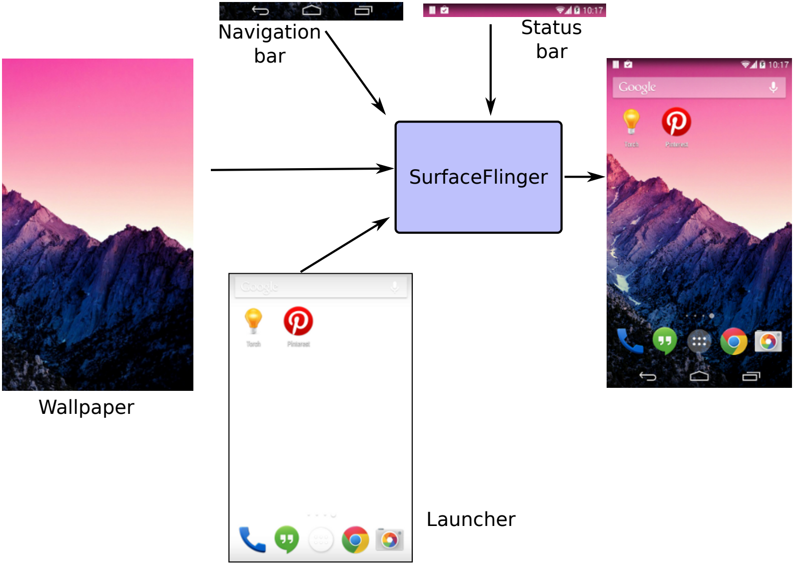
3D rendering path

- An activity can instead create a *GLSurfaceView* and use OpenGL ES bindings for Java (the android.opengl.* classes)
- Using either the vendor GPU driver (which must support OpenGL ES 2.0 and optinally 3.0)
- Or as a fall-back, using PixelFlinger, a software GPU that implements OpenGL ES 1.0 only
- Once again, the drawing is rendered to a Surface

3D rendering path



Composition

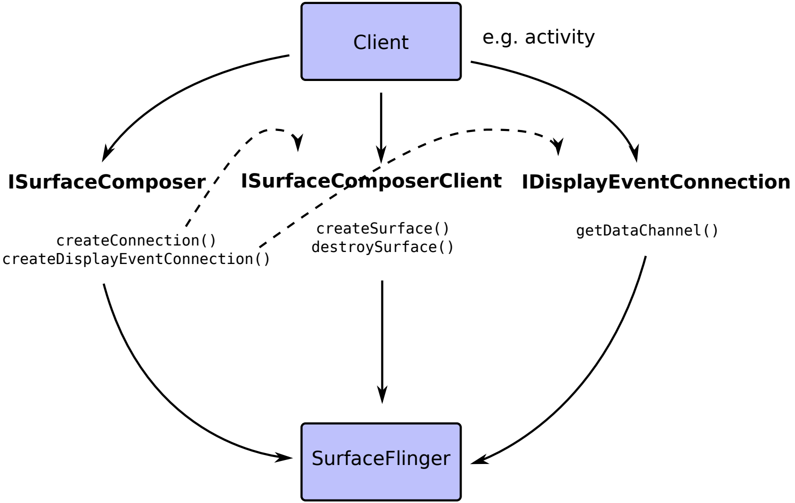


SurfaceFlinger

`frameworks/native/services/surfaceflinger`

- A high-priority native (C++) daemon, started by init with UID=*system*
- Services connections from activities via Binder interface *ISurfaceComposer*
- Receives activity status from Activity Manager
- Receives window status (visibility, Z-order) from Window Manager
- Composits multiple Surfaces into a single image
- Passes image to one or more displays
- Manages buffer allocation, synchronisation

SurfaceFlinger binder interfaces



ISurfaceComposer

- ISurfaceComposer
  - Clients use this interface to set up a connection with SurfaceFlinger
  - Client begins by calling *createConnection()* which spawns an ISurfaceComposerClient
  - Client calls *createGraphicBufferAlloc()* to create an instance of IGraphicBufferAlloc (discussed later)
  - Client calls *createDisplayEventConnection()* to create an instance of IDisplayEventConnection
  - Other methods include *captureScreen()* and *setTransactionState()*

ISurfaceComposerClient

- ISurfaceComposerClient
  - This interface has two methods:
  - *createSurface()* asks SufraceFlinger to create a new Surface
  - *destroySurface()* destroys a Surface

IDisplayEventConnection

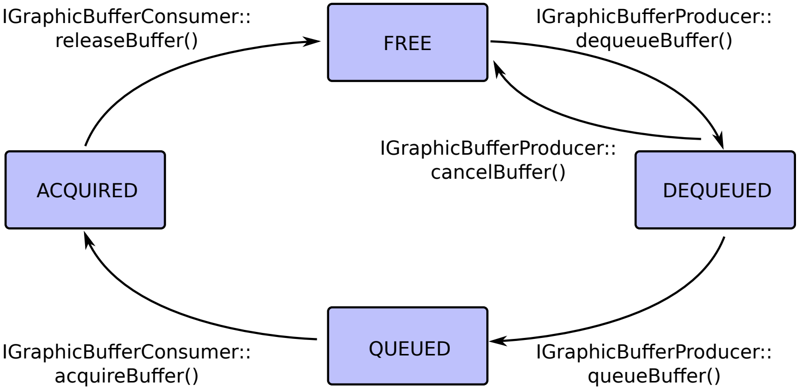
- IDisplayEventConnection
  - This interface passes vsync event information from SurfaceFlinger to the client
  - *setVsyncRate()* sets the vsync event delivery rate: value of 1 returns all events, 0 returns none
  - *requestNextVsync()* schedules the next vsync event: has no effect if the vsync rate is non zero
  - *getDataChannel()* returns a BitTube which can be used to receive events

BufferQueue

`frameworks/native/include/gui/BufferQueue.h`

- Mechanism for passing GraphicBuffers to SurfaceFlinger
- Contains an array of between 2 and 32 GraphicBuffers
- Uses interface *IGraphicBufferAlloc* to allocate buffers (see later)
- Provides two Binder interfaces
  - *IGraphicBufferProducer* for the client (Activity)
  - *IGraphicBufferConsumer* for the consumer (SurfaceFlinger)
- Buffers cycle between producer and consumer

BufferQueue state diagram



BufferQueue

- Default number of buffer slots since JB is 3 (previously 2)
  - In JB you can compile Layer.cpp with `TARGET_DISABLE_TRIPLE_BUFFERING` to return to 2 slots
- Call *setBufferCount()* to change the number of slots
- BufferQueue operates in two modes:
  - Synchronous: client blocks until there is a free slot
  - Asynchronous: queueBuffer() discards any existing buffers in QUEUED state so the queue only holds the most recent frame

GraphicBuffer

`frameworks/native/include/ui/GraphicBuffer.h`

- Represents a buffer, wraps ANativeWindowBuffer
- Attributes including *width, height, format, usage* inherited from ANativeWindowBuffer

Composition

- On a vsync event, SurfaceFlinger calls *handleMessageRefresh()* which goes through a composition cycle:
  - *preComposition()*: sort layers by Z order and call onPreComposition() for each
  - *doComposition()*: loop through displays: if there is a dirty region, mark it to be drawn then call *postFameBuffer()* to do the drawing
  - *postComposition()*: loop through layers in Z order and call *onPostComposition()*

Layer

`frameworks/native/services/surfaceflinger/Layer.h`

- Each Layer has
  - Z order
  - Alpha value from 0 to 255
  - visibleRegion
  - crop region
  - transformation: rotate 0, 90, 180, 270: flip H, V: scale
- SurfaceFlinger composits the layers using
  - HWComposer, if it supports the operation
  - Fall back to the GPU, via OpenGL ES (version 1.0 only, for historical reasons)

HWComposer

`hardware/libhardware/include/hardware/hwcomposer.h`

- HWComposer is a vendor-supplied library, at run-time in /system/lib/hw/hwcomposer.[product name].so
- Optional: in all cases there are fall-backs if HWC is absent
- HWC does several different things
  - sync framework (vsync callback)
  - modesetting, display hotplug (e.g. hdmi)
  - compositing layers together using features of the display controller
  - displaying frames on the screen

prepare() and set()

- SurfaceFlinger calls HWComposer in two stages
- *prepare()*
  - Passes a list of layers
  - For each layer, HWComposer returns
  - `HWC_FRAMEBUFFER`: SurfaceFlinger should write this layer (using OpenGL)
  - `HWC_OVERLAY`: will be composed by HWComposer
- *set()*
  - Passes the list of layers for HWComposer to handle
- set() is used in place of eglSwapBuffers()

vsync

- Since JB 4.1 SurfaceFlinger is synchronised to a 60Hz (16.7ms period) vsync event
- If HWComposer present, it is responsible for vsync
  - Usually using an interrupt from the display: if no h/w trigger, fake in software
  - *vsync()* is a callback registered with HWComposer
  - Each callback includes a display identifier and a timestamp (in ns)
- If no HWComposer, SurfaceFlinger uses 16ms timeout in s/w

Displays

- HWComposer defines three display types

```
HWC_DISPLAY_PRIMARY    e.g. built-in LCD screen
HWC_DISPLAY_EXTERNAL   e.g. HDMI, WiDi
HWC_DISPLAY_VIRTUAL    not a real display
```

- For each display there is an instance of *DisplayDevice* in SurfaceFlinger

IGraphicBufferAlloc and friends

`frameworks/native/include/gui/IGraphicBufferAlloc.h`

- Binder interface used by SurfaceFlinger to allocate buffers
- Has one function *createGraphicBuffer*
- Implemented by class GraphicBufferAllocator, which wraps the ANativeWindowBuffer class
- Uses Gralloc.alloc to the the actual allocation
- Underlying buffer is referenced by a *buffer_handle_t* which is a file descriptor (returned by gralloc alloc)
- Binder can pass open file descriptors from process to process
- Access buffer data using *mmap*

Buffer usage and pixel format

`frameworks/native/include/ui/GraphicBuffer.h`

```
USAGE_HW_TEXTURE           OpenGL ES texture
USAGE_HW_RENDER            OpenGL ES render target
USAGE_HW_2D                2D hardware blitter
USAGE_HW_COMPOSER          used by the HWComposer HAL
USAGE_HW_VIDEO_ENCODER     HW video encoder
```

`frameworks/native/include/ui/PixelFormat.h`

```
PIXEL_FORMAT_RGBA_8888      4x8-bit RGBA
PIXEL_FORMAT_RGBX_8888      4x8-bit RGB0
PIXEL_FORMAT_RGB_888        3x8-bit RGB
PIXEL_FORMAT_RGB_565        16-bit RGB
PIXEL_FORMAT_BGRA_8888      4x8-bit BGRA
```

Gralloc

`hardware/libhardware/include/hardware/gralloc.h`

- Gralloc is a vendor-supplied library, at run-time in /system/lib/hw/gralloc.[product name].so
- Does two things
  - gralloc alloc: allocates graphic buffers
  - gralloc framebuffer: interface to Linux framebuffer device, e.g. */dev/graphics/fb0*
- gralloc alloc allocates all graphic buffers using a kernel memory manager, typically ION
- Selects appropriate ION heap based on the buffer usage flags

OpenGL ES

- The Khronos *OpenGL ES* and *EGL* APIs are implemented in these libraries
  - `/system/lib/libEGL.so`
  - `/system/lib/libGLESv1_CM.so`
  - `/system/lib/libGLESv2.so`
  - `/system/lib/libGLESv3.so` (optional from JB 4.3 onwards: actually a symlink to libGLESv2.so)
- In most cases they simply call down to the vendor-supplied libraries in `/system/lib/egl`

EGL

- EGL is the Khronos *Native Platform Graphics Interface*
- Rendering operations are executed in an EGLContext
- In most cases the EGLContext is based on the default display
- The mapping from the EGL generic display type is done in

  `frameworks/native/opengl/include/EGL/eglplatform.h`

```
typedef struct ANativeWindow*   EGLNativeWindowType;
```

- *EGLNativeWindowType* is defined in

  `system/core/include/system/window.h`

OpenGL vendor implementation

- The vendor OpenGL libraries form the interface to the GPU
- Responsible for
  - creating display lists
  - scheduling work for the GPU
  - managing buffer synchronisation (typically using fences, see background at the end)
- Usually there is a kernel driver which handles low level memory management, DMA and interrupts
- The kernel interface is usually a group of ioctl functions

- Questions?

Background: fences

Buffer synchronisation

- There are many producers and consumers of graphics buffers
- Pre JB sync was *implicit*: buffer not released until operation complete
- Did not encourage parallel processing
- JB introduced *explicit* sync: each buffer has a sync object called a *fence*
- Means a buffer can be passed to the next user before operations complete
- The next user waits on the fence before accessing the buffer contents

Synchronisation using fences

- Represented by file handles: can be passed between applications in binder messages
- Can also be passed from applications to drivers
- Each device driver (display, camera, video codec...) has its own *timeline*
- A fence may have synchronisation points on multiple timelines
- Allows buffers to be passed between multiple devices

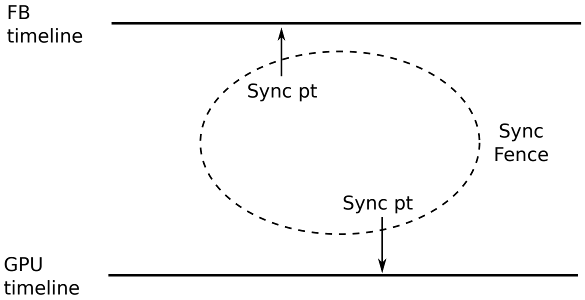
Timeline and sync point

- Timeline
    - Per-device (display, GPU, camera, ...)
    - Monotonically increasing 32-bit value
    - Incremented after each event (essentially it is a count of the jobs processed by the device)
- Sync point
    - A point on a timeline
    - Becomes *signalled* when the timeline passes it

Fence

- Fence
  - A collection of one or more sync points, possibly from different timelines
  - Represented by a file descriptor so an application can wait using poll()
  - Two fences can be merged to create a new fence that depends on all the sync points of the original pair

Fence: example



Background: ION

Memory constraints

- Often necessary for a buffer to be accessed by hardware
- Example: graphics buffer and display controller or GPU
- Hardware may constrain memory access
- Example: hardware without IOMMU usually needs physically contiguous memory
- To avoid copying, the memory must be allocated for the most constrained device

ION

- Previous memory allocators include pmem (Qualcomm), cmem (TI), and nvmap (NVIDA)
- ION provides a unified interface for these needs
  - Different allocation constraints
  - Different caching requirements
  - But the programmer still has to make the right choices

Types of heap

- ION_HEAP_TYPE_SYSTEM
    - memory allocated via vmalloc
- ION_HEAP_TYPE_SYSTEM_CONTIG
    - memory allocated via kmalloc
- ION_HEAP_TYPE_CARVEOUT
    - memory allocated from a pre reserved carveout heap
    - allocations are physically contiguous

Heap flags

- `ION_FLAG_CACHED`
  - mappings of this buffer should be cached, ION will do cache maintenance when the buffer is mapped for DMA
- `ION_FLAG_CACHED_NEEDS_SYNC`
  - Cache must be managed manually, e.g. using `ION_IOC_SYNC`

