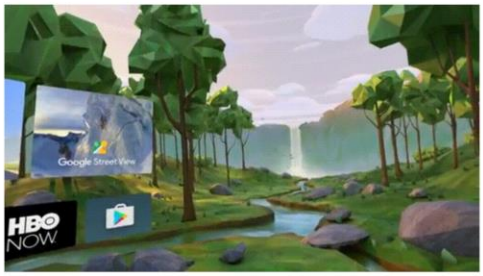

GENERATIONS / VANCOUVER
SIGGRAPH2018
Preparing Android for XR
Jiwen Cai, Google
© 2018 SIGGRAPH. All Rights Reserved

Photography & Recording Encouraged

Preparing Android for XR

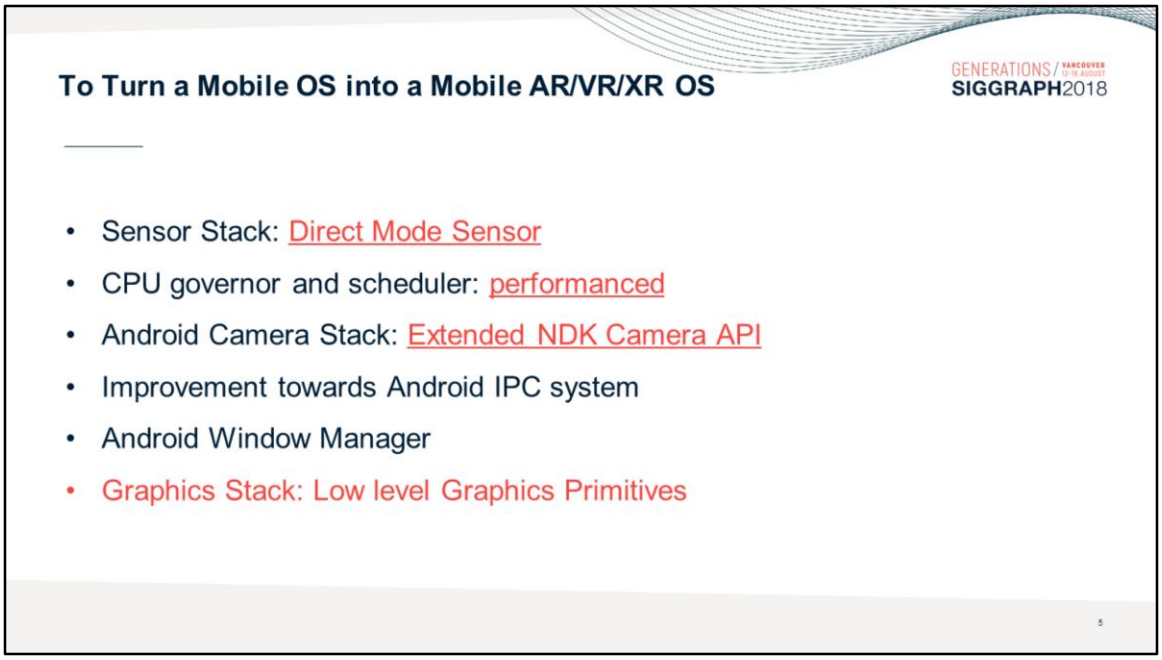
Jiwen Cai, Google

To Turn a Phone into a XR Device



Started Cardboard

- Built on Valina Android
- 100ms+ Mo2Pho
- Android wasn't originally designed for AR and VR applications

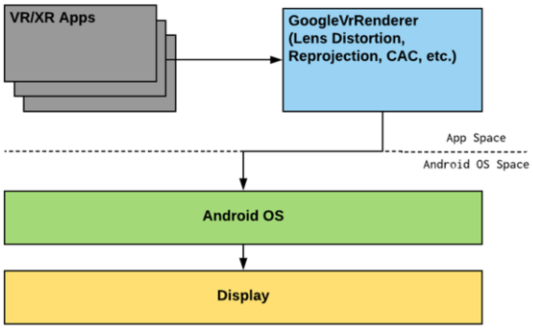


We improved a tons of Android sub-systems: from sensor, to thread scheduler, from IPC mechanism to window manager.

But since this is SIGGRAPH, I guess it's better to focus on graphics.

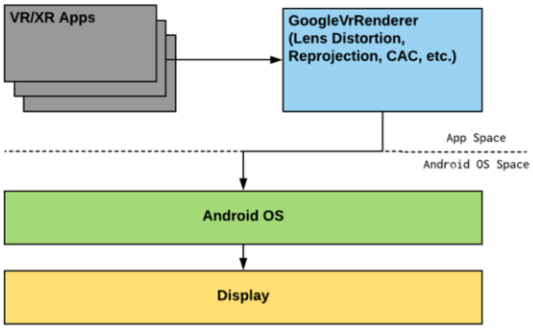
Google VR/XR Graphics Stack: Three “Simple” Steps

- Application Draws a Frame in stereo
- **GoogleVrRenderer** applies VR specific treatments
- Android sends the output of the renderer to the display hardware



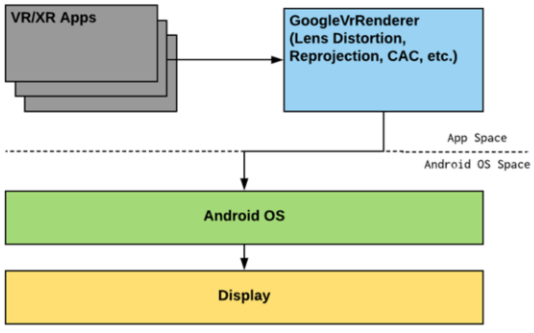
There Is a Big Problem

- The big Android block sitting in between **GoogleVrRenderer** and the display is in the way
- There is at least 50 ms of latency introduced just by the graphics stack



Three Steps to Improve Android

- Step 1: Understanding the existing Android graphics stack
- Step 2: Punching a hole through Android: single buffer rendering
- Step 3: Doing it right: low level Android graphics primitives



Step 1: Starting from GLSurfaceView.Renderer

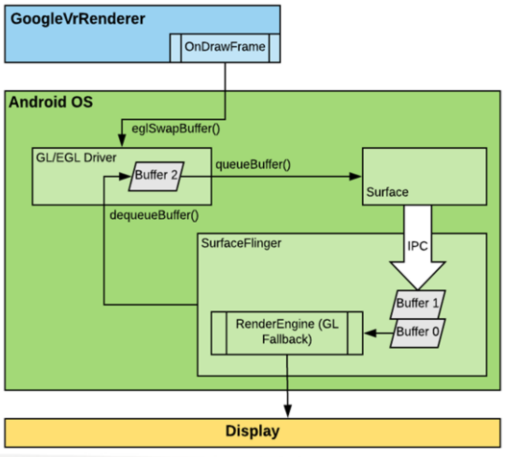

```
import android.opengl.GLES20;
import android.opengl.GLSurfaceView;

class GoogleVrRenderer implements GLSurfaceView.Renderer {
 @Override
  public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    GLES20.glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
  }

  @Override
  void onDrawFrame(GL10 gl) {
     gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
     //... other gl calls to render the scene ...
  }
}
```

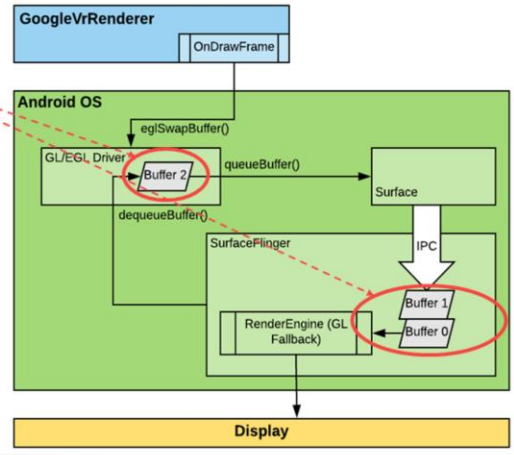
- Hey, that's simple! But maybe too simple if you care about latency.
- What is a **Surface**?
- What happens after **onDrawFrame**?
- Why my frame doesn't show up on display until 50ms ~ 100ms later?

What's under the Hood?



What's under the Hood?

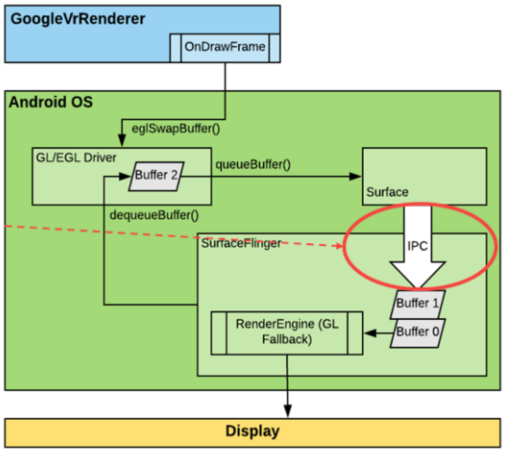
Android graphics is triple buffered. There was no easy way to turn that off.



What's under the Hood?

Android graphics is triple buffered. There was no easy way to turn that off.

IPC hop between GoogleVrRenderer and SurfaceFlinger was expensive and unpredictable.

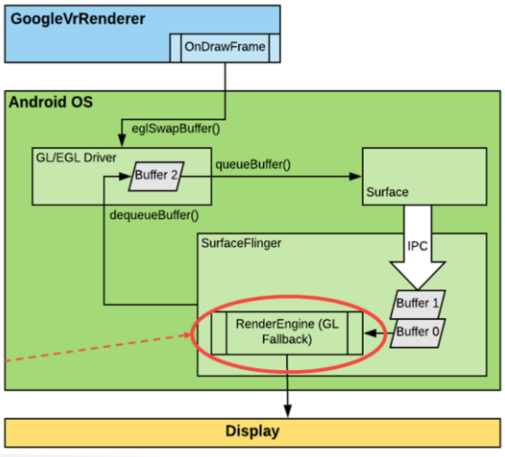


What's under the Hood?

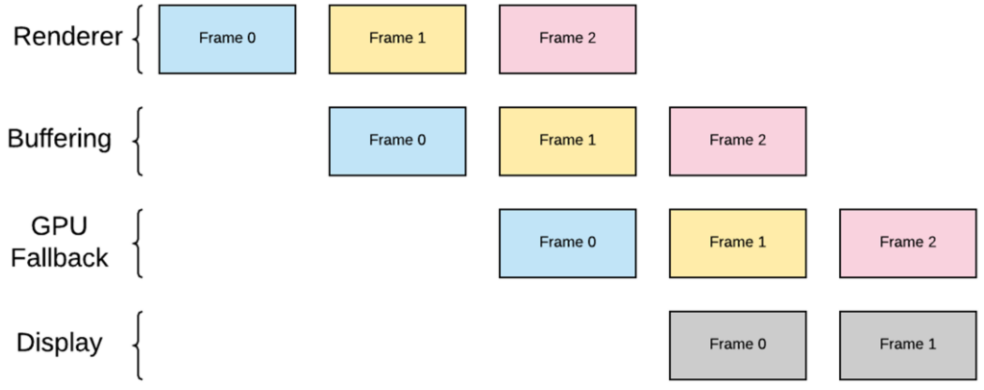
Android graphics is triple buffered. There was no easy way to turn that off.

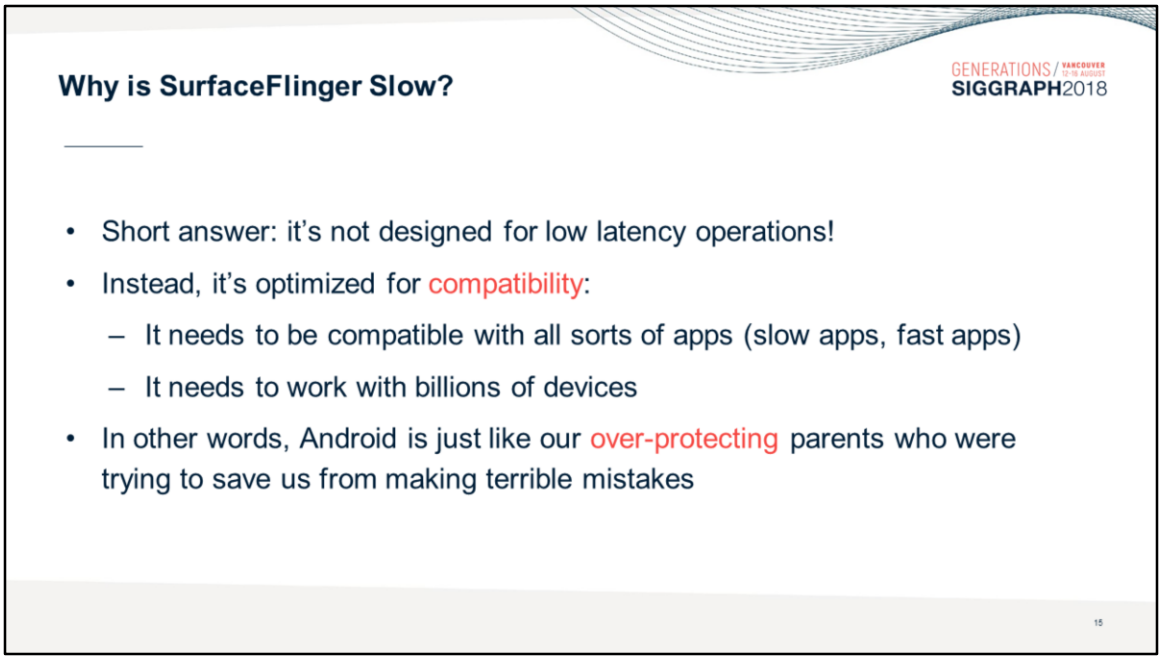
IPC hop between GoogleVrRenderer and SurfaceFlinger was expensive and unpredictable.

SurfaceFlinger has a GPU backend and it may wait for its GL call to finish before sending its output buffer to display.

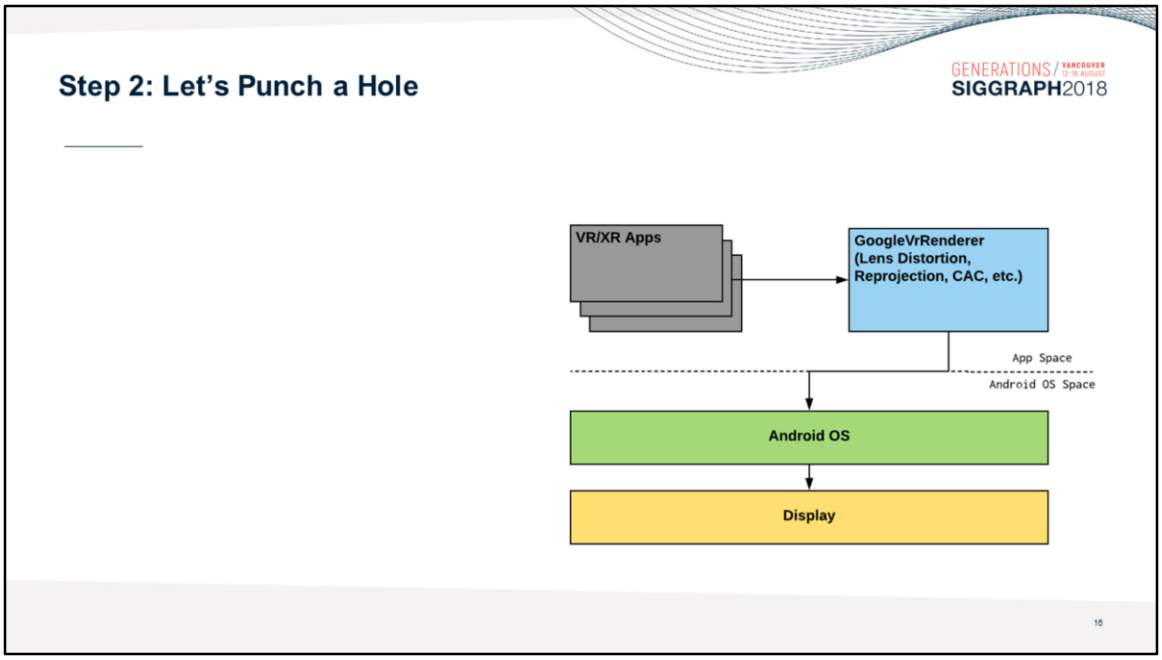


A Very Long Pipeline





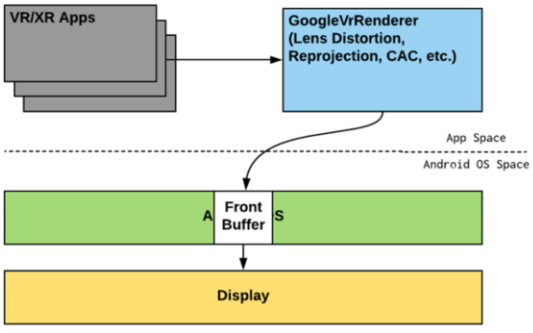
- Not sure about you guys, but it definitely sounds like my Mom when I was 15.
- What if you cannot render frame in time? Please do triple buffering.
- What if your render loop crash? Please draw in a different process.
- What if your hardware doesn't support hardware compositing? Let's wait for GL fallback.



Now, we understand why Android graphics pipeline introduces huge latencies, just like our over protecting mom. What can we do in the meantime? Just like most teanagers, we tried to workaround the system our parents set up for us.

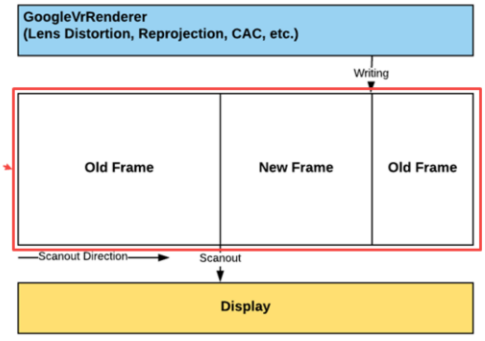
Step 2: Let's Punch a Hole

- There is a big pipeline sitting in the way, let's work around it
- Instead of swapping buffers every frame, just share the same buffer between the renderer and the display



Reading from and Writing into the Same Buffer

Render into the front buffer directly. The buffer is in landscape mode and scanout line by line from left to right.

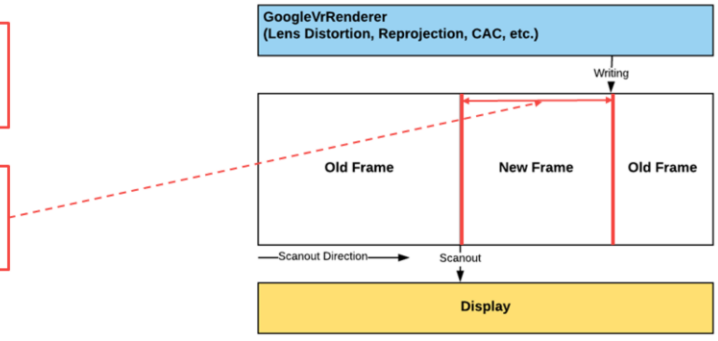


John Carmack's "Creative Way to Cheat" Talk with UMKC-SCE @ 29:41

Reading from and Writing into the Same Buffer

Render into the front buffer directly. The buffer is in landscape mode and scanout line by line from left to right.

Latency is between the line being scanned out and the line renderer is writing into. In practice, we draw into stripes.



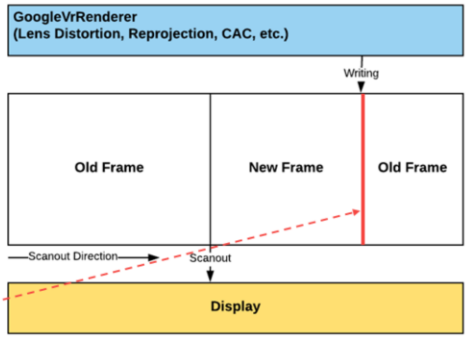
John Carmack's "Creative Way to Cheat" Talk with UMKC-SCE @ 29:41

Reading from and Writing into the Same Buffer

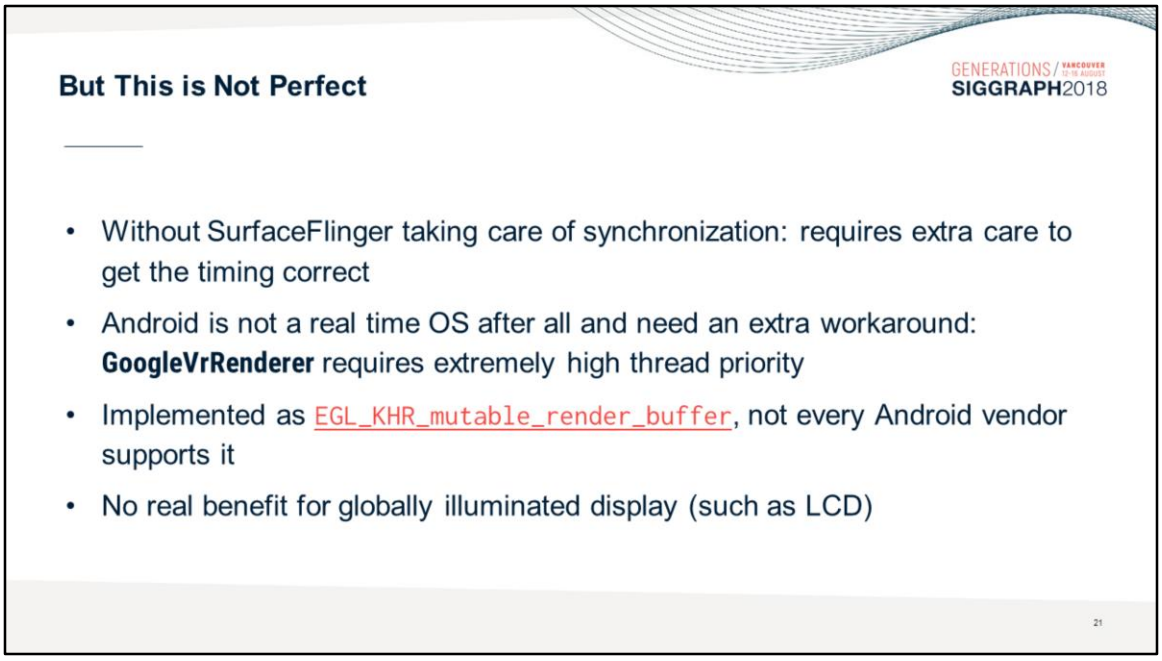
Render into the front buffer directly. The buffer is in landscape mode and scanout line by line from left to right.

Latency is between the line being scanned out and the line renderer is writing into. In practice, we draw into stripes.

Racing the scanline: always render ahead of the scanline. Otherwise there will be tearing.



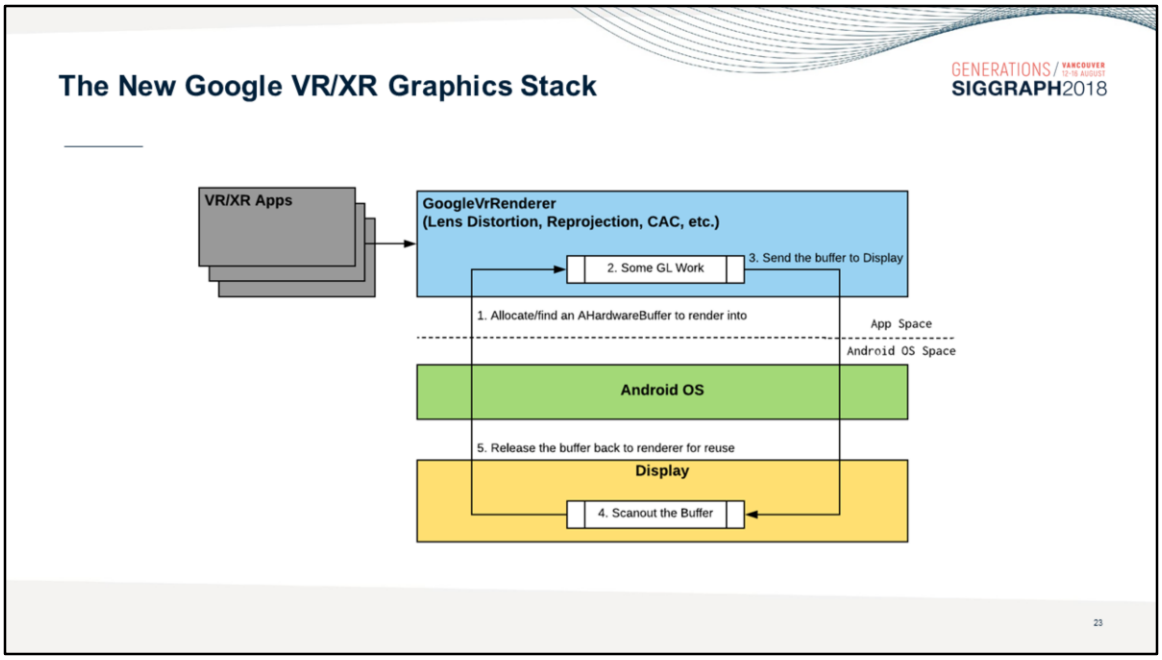
John Carmack's "Creative Way to Cheat" Talk with UMKC-SCE @ 29:41



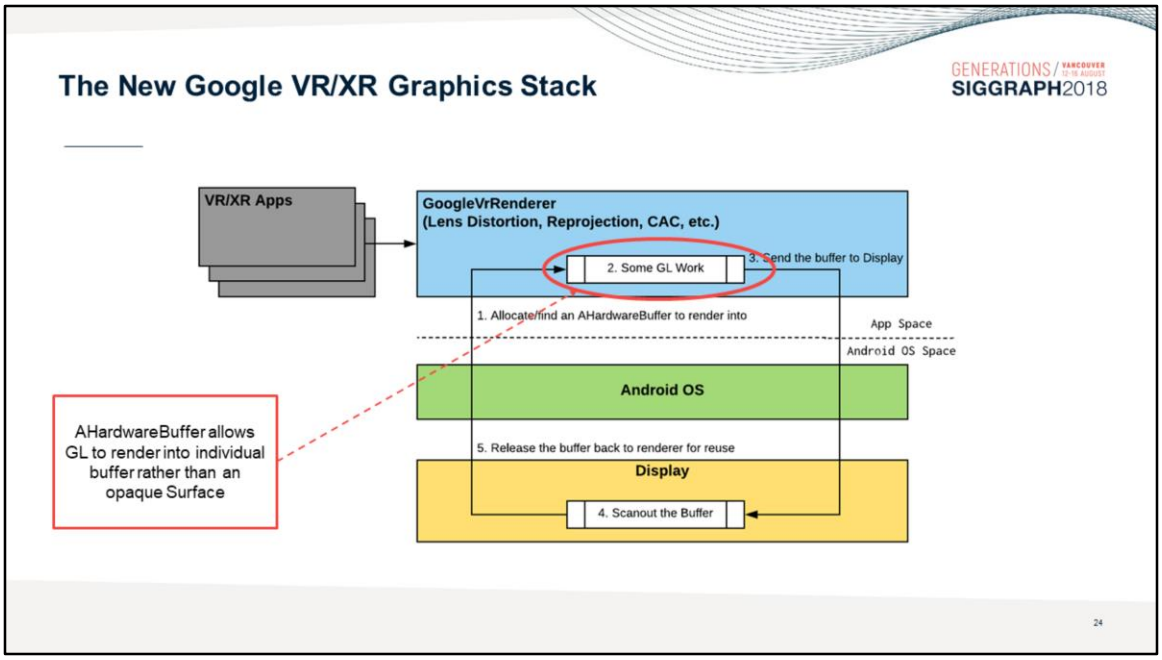
The priority is so high that I'm not allowed to publicly talking about it.

Step 3: Low Level Graphics Primitives

- Exposed Graphics Primitive: AHardwareBuffer
  - GL/EGL can sample from and/or render into an individual buffer
- Built an BufferQueue alternative: BufferHub
  - IPC-agnostic buffer sharing
  - IPC-free reader/writer state signalling
- Grant direct display hardware control to **GoogleVrRenderer**: VrFlinger
  - Precise control of the buffer to be synced out at any vsync



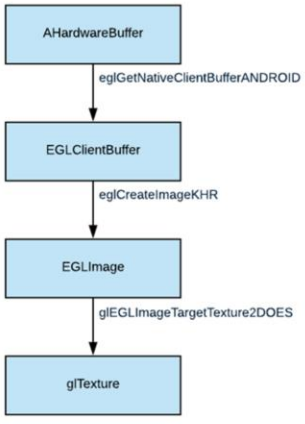
Single buffer can still be supported to achieve sub-frame latency.

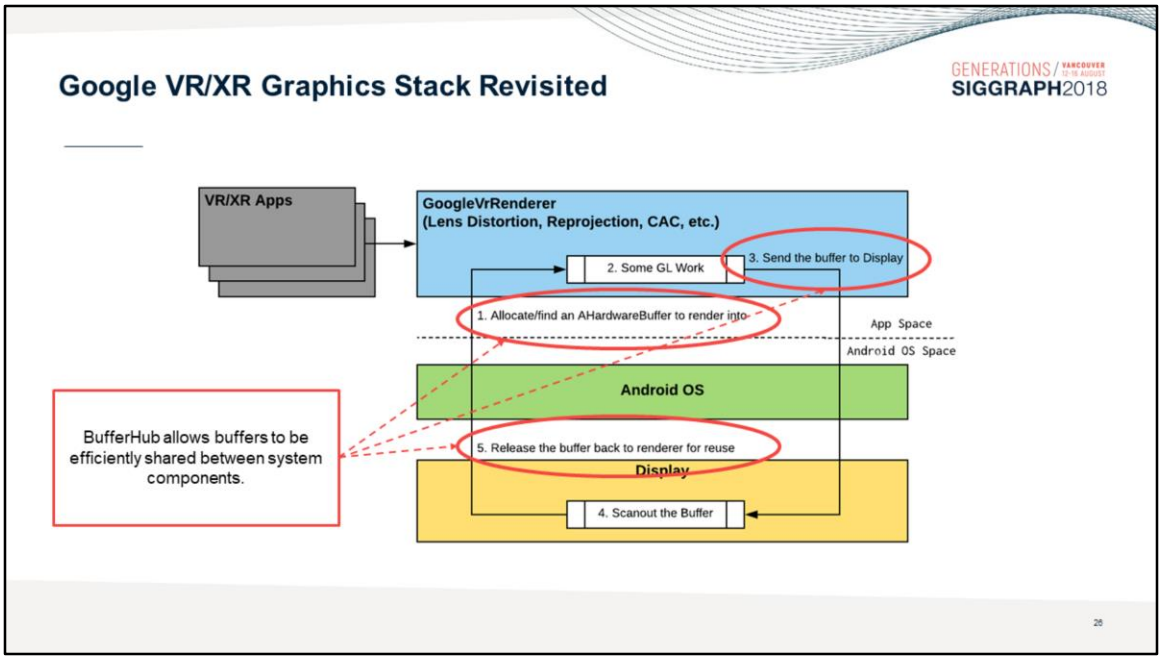


Single buffer can still be supported to achieve sub-frame latency.

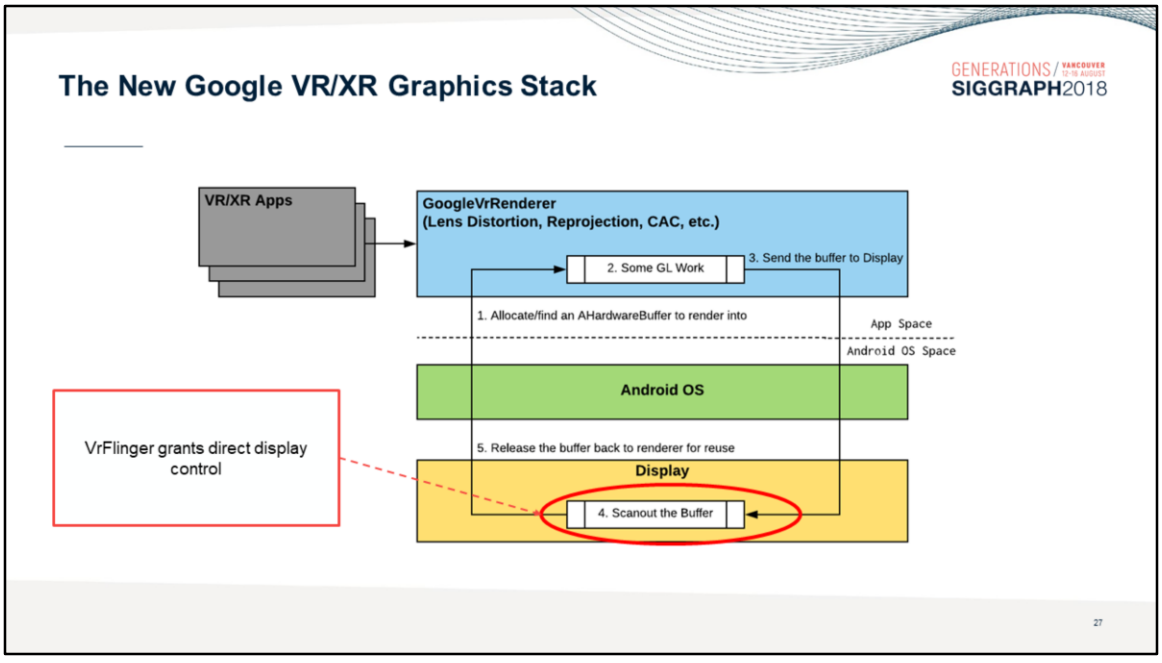
Render into a buffer rather than a Surface


```
class NativeGoogleVrRenderer {
 // Render into an AHardwareBuffer.
 void drawFrame(AHardwareBuffer* buffer, int* out_fence_fd) {
  // Map the AHardwareBuffer to the frame buffer.
  EGLClientBuffer eglBuffer = eglGetNativeClientBufferANDROID();
  EGLImageKHR eglImage = eglCreateImageKHR(..., eglBuffer, ...);
  glBindTexture(GL_TEXTURE_2D, texture_id_);
  glEGLImageTargetTexture2DOES(GL_TEXTURE_2D, eglImage);
  glBindFramebuffer(GL_FRAMEBUFFER, framebuffer_id_);
  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0
                         GL_TEXTURE_2D, texture_id_, 0);
  // ... gl calls to render the scene ...
  // ... out_fence_fd will be signaled once all GPU work is done ...
 }
};
```





Single buffer can still be supported to achieve sub-frame latency.



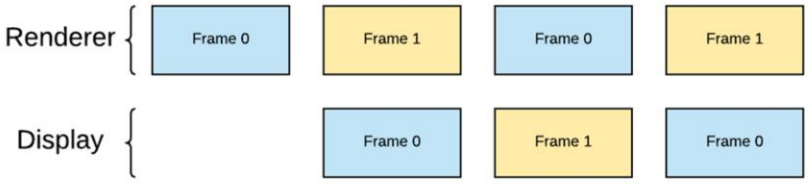
Single buffer can still be supported to achieve sub-frame latency.

Revisit: GoogleVrRenderer


```
class NativeGoogleVrRenderer {
 void renderLoop() {
  while (1) {
   // Allocate or reused a buffer.
   AHardwareBuffer* buffer = dvrWaitForBuffer();
   int fence = -1;
   drawFrame(buffer, &fence);
   // Mutate buffer state in shared memory rather than explicit IPC.
   AHardwareBuffer_post(buffer);
   // A transaction towards SurfaceFlinger indicating the buffer to be
   // used for the VSYNC.
   dvrPostBuffer(buffer, fence);
  }
 }
};
```

- The render loop can be written in C++, no JNI needed
- The renderer now owns the entire render loop
- The buffer can be posted to SurfaceFlinger right after all CPU calls

A Shorter Pipeline



Revisit: Shared Buffer Mode


```
class NativeGoogleVrRenderer {
 void renderLoop() {
  AHardwareBuffer* buffer = dvrWaitForBuffer();
  AHardwareBuffer_post(buffer);
  dvrPostBuffer(buffer, /*fence=*/-1);

  while (1) {
   // Some care logic to timing.
   timingTheVsync();
   drawFrame(buffer);
  }
 }
};
```

- Simply post the buffer only once.
- No fence required, but need perfect timing for the draw call.

Recap

- Android wasn't ideal for VR/AR because its graphics API was too restrictive
- We bypassed the entire Android graphics stack with front buffer rendering
- We introduced new low level graphics primitives to explicitly control our render loop

Future Work

- Vulkan support for AHardwareBuffer
- Bring BufferHub into mainline Android with public SDK/NDK API
- Making SurfaceFlinger more flexible and configurable

Thank You!

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Questions?

33