What’s new in Camera Development ?

Creating delightful camera experiences with the latest features in Android through Camera2 and CameraX

Jayant Chowdhary
Jag Saund
Kailiang Chen

Prerequisites for this talk

- Basics of Android app development
- Basics of Camera2 and CameraX APIs
    - We assume that you know how to create a basic camera app
- We’ll provide links to resources wherever applicable

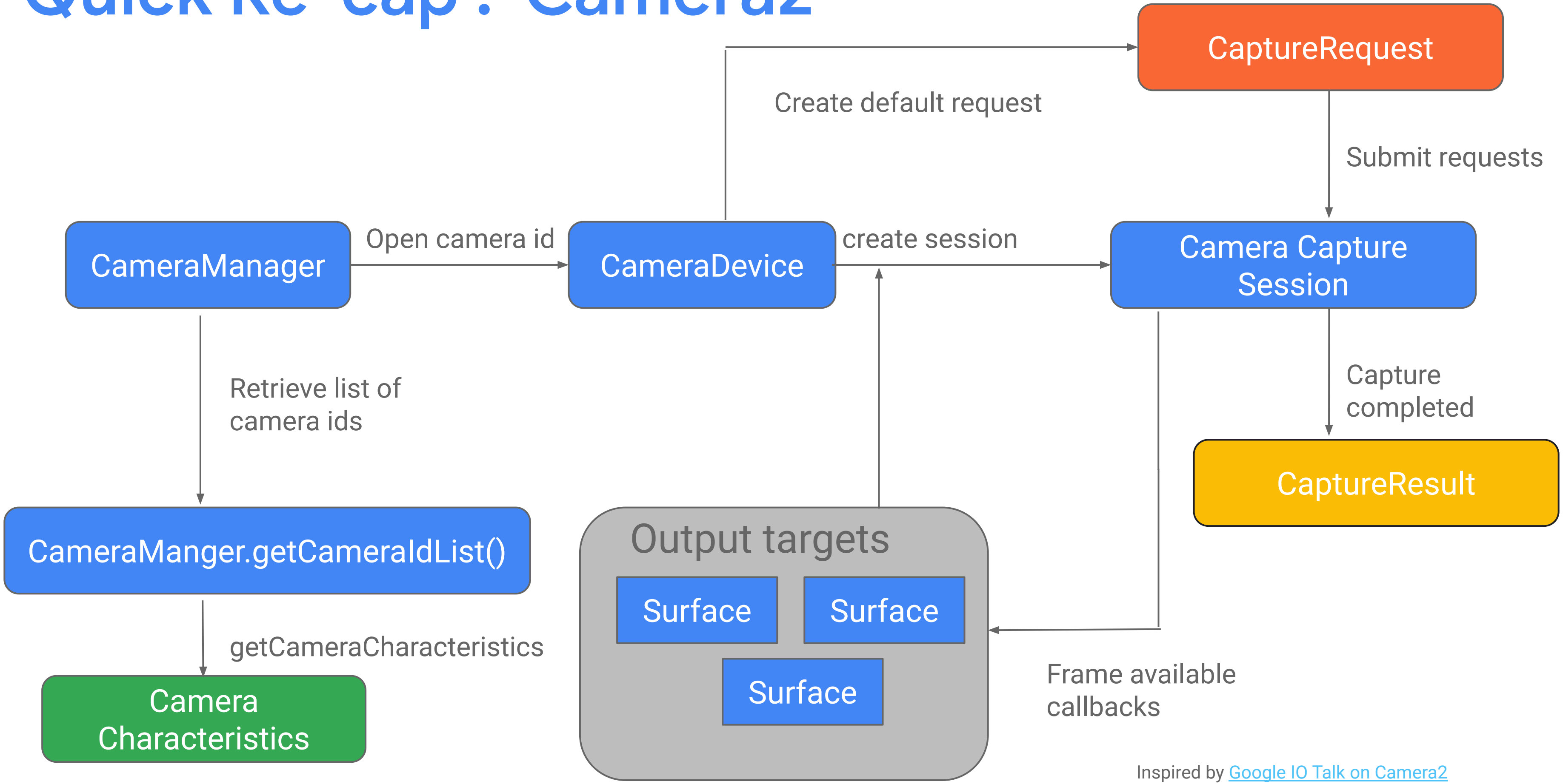
Agenda

- Quick re-cap of Camera2 and CameraX
- CameraX vs Camera2: Which API to use
- Working towards building a camera app using the latest features in Camera2 and CameraX
    - Enhance Expressibility
        - Preview Stabilization
        - Stream Use Cases
        - Concurrent Cameras
    - Elevate your Captures
        - Night Mode Camera Extensions
        - HDR Video and Ultra HDR Capture

Quick Re-cap : Camera2

- Introduced in 2014 - Android Lollipop
- Aimed at super charging camera development on Android
- Models the camera API as a `pipeline model` and exposes many details of the internal camera system
- Really powerful and gives apps a ton of control

Quick Re-cap : Camera2



Inspired by [Google IO Talk on Camera2]

Google

Simple Camera2 App with Preview

- Handle Asynchronous calls : CameraManager.openCamera(), `CameraDevice`.createCaptureSession()
- Compute view rotation /mirroring to use , preview resolution and aspect ratio to configure for `CameraDevice` - which can get tricky on foldables
- Manage camera open and close with lifecycle owner
- Simple app with preview takes some non trivial amount of code to implement
- Camera2 sample

Quick Re-cap : CameraX

- Camera2: powerful but complex
- CameraX: faster development
  - Built on top of Camera2 but hides the details
  - Offers advanced capabilities with simplicity
  - Broad compatibility – 98% of existing Android devices
  - Faster version releases
  - Production apps such as YT shorts use CameraX

Quick Re-cap : CameraX Use cases

Preview

ImageCapture

VideoCapture

ImageAnalysis

- Configure use case with desired options
- Tell Android what to do with outputs by attaching listeners
- Bind use case to life cycles

CameraX example: App with only Preview

```
class MainActivity : ComponentActivity() {
    private lateinit var cameraController: LifecycleCameraController
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ...
        startCamera()
    }

    private fun startCamera() {
        val previewView: PreviewView = viewBinding.viewFinder
        cameraController = LifecycleCameraController(baseContext)
        cameraController.bindToLifecycle(this)
        cameraController.cameraSelector = CameraSelector.DEFAULT_FRONT_CAMERA
        previewView.controller = cameraController
    }
```

< 15 lines of camera code to get preview running with tap to focus and pinch to zoom

Which API do I use : Camera2 or CameraX ?

- In general, we **recommend using CameraX**, it'll simplify your camera development quite a bit, especially if you're using Camera1

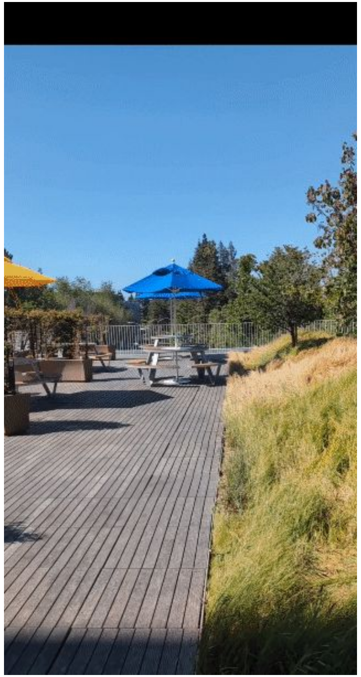
Camera2	**CameraX**
Complex low-level control	Simplified high-level control
High performance, granular control	Optimized for ease of use and performance
Full manual controls over camera	Access to most camera features
Developer velocity is slower with more boilerplate code	Developer velocity is faster, less boilerplate code
Good for complex custom applications	Good for common camera applications

Enhance Expressibility

Preview Stabilization

Preview Stabilization

- Preview set up
- Using the same stream for preview and recording.
- Current recording appears shaky



Google

Preview Stabilization

- Shaky video
- We can do better right?
- Camera2 API CaptureRequest control :
  CONTROL_VIDEO_STABILIZATION_MODE
  - OFF - No stabilization
  - ON
  - **PREVIEW_STABILIZATION (API Level 31)**

Preview Stabilization

- VIDEO_STABILIZATION: ON
  - 1080p streams (MediaCodec / MediaRecorder) stabilized
  - FoV reduction (WYS != WYG)
  - Prioritize stabilization quality and file recording over real-timeness

Preview stream unstabilized full FoV

Record stream stabilized with FoV reduction

Images generated by Gemini

Preview Stabilization

- VIDEO_STABILIZATION: PREVIEW_STABILIZATION guarantees non-RAW streams are stabilized with the same quality of stabilization. WYSIWYG.
- FoV reduction limited to 20% on both horizontal and vertical crop
- All non RAW streams have the same FoV
- Optimizes for ‘real-timeness’

Preview stream stabilized with x% FoV reduction

Record stream stabilized with x% FoV reduction

Images generated by Gemini

Preview Stabilization

- Limitations on Preview Stabilization
  - due to real time computation per frame
- Guaranteed stream combinations:
  - 1440p preview stream + maximum size YUV / JPEG capture
  - 1440p YUV / PRIV stream + 1080p YUV / PRIV
  - Details can be found here

Camera2: Preview Stabilization in code

```
private fun configurePreviewCaptureRequest(previewSurface: Surface): CaptureRequest.Builder
= camera.createCaptureRequest(
        CameraDevice.TEMPLATE_PREVIEW
    ).apply {
        addTarget(previewSurface)
    }
```

Camera2: Preview Stabilization in code

```
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
private fun configurePreviewCaptureRequest(previewSurface: Surface): CaptureRequest.Builder =
    camera.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW).apply {
        addTarget(previewSurface)
        val isPreviewStabilizationAvailable =
            characteristics.get(CONTROL_AVAILABLE_VIDEO_STABILIZATION_MODES)
            ?.contains(CONTROL_VIDEO_STABILIZATION_MODE_PREVIEW_STABILIZATION) ?: false
        if (isPreviewStabilizationAvailable) {
            set(CONTROL_VIDEO_STABILIZATION_MODE,
                CONTROL_VIDEO_STABILIZATION_MODE_PREVIEW_STABILIZATION)
        }
    }
```

CameraX: Preview Stabilization in code

```kotlin
private fun isPreviewStabilizationSupported(
    cameraProvider: CameraProvider,
    lensFacing: Int
): Boolean {
    val cameraInfos = cameraProvider.availableCameraInfos
    for (cameraInfo in cameraInfos) {
        if (cameraInfo.lensFacing == lensFacing) {
            return Preview.getPreviewCapabilities((cameraInfo)).isStabilizationSupported
        }
    }
    return false
}
```

CameraX: Preview Stabilization in code

```
private fun isPreviewStabilizationSupported(...): Boolean { ... }

private suspend fun startCamera() {
    val cameraProvider = ProcessCameraProvider.getInstance(this).await()
    val previewBuilder = Preview.Builder()
    val cameraSelector = CameraSelector.DEFAULT_FRONT_CAMERA;
    if (isPreviewStabilizationSupported(cameraProvider, CameraSelector.LENS_FACING_FRONT)) {
        previewBuilder.setPreviewStabilizationEnabled(true)
    }
    val preview = previewBuilder.build()
    preview.setSurfaceProvider(viewFinder.surfaceProvider)
    // set up remaining use cases
}
```

Stream Use Cases

Google

Optimizing Camera Streams for your Use Case

- Now you have an app where your preview isn’t as shaky, all your streams have the same field of view, so they’re consistent.
- You want to introduce more features in your app - for instance say video calling. Video calls can be long...and keeping the camera on for long means lots of power consumption
- Do we have a way of telling the system - optimize power even if it means reducing image quality a bit ?

Make your intentions clear : Stream Use Cases

- Example: YUV stream being used for preview or still capture ? The camera sub-system doesn't know so YUV image quality could be worse than corresponding JPEG image quality for the same image size.
- In many cases it's unclear what purpose a stream is being used for.

Stream Use Cases

- Give the camera sub-system hints about what a stream is actually going to be used for
- Attach a ‘use case’ to a camera2 `OutputConfiguration` using `OutputConfiguration.setStreamUseCase()`
- Introduced in API level 33

Available Stream Use Cases

DEFAULT
Covers existing / default behavior for all streams. No hints are given to the camera sub-system.

PREVIEW
Optimized for performance and usability as a viewfinder, but not necessarily for image quality.

VIDEO_CALL
Long-running video call optimized for both power efficiency and video quality. Image quality may be reduced as a tradeoff.

Available Stream Use Cases

VIDEO_RECORD

Optimized for high-quality video capture, including high-quality image stabilization if supported by the device and enabled by the application

STILL CAPTURE

Optimized for high-quality high-resolution capture, and not expected to maintain preview-like frame rates.

PREVIEW_VIDEO_STILL

Single stream for combined purposes of preview, video, and still capture. Camera device aims to make the best tradeoff between individual use cases.

Camera2 Example: VIDEO_CALL Stream Use Case

```kotlin
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun createCaptureSession(
    device: CameraDevice,
    targets: List<Surface>,
    handler: Handler? = null
): CameraCaptureSession {
    // Session configuration code
}
```

Camera2 Example: VIDEO_CALL Stream Use Case

```
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
private fun createCaptureSession(
   device: CameraDevice,
   targets: List<Surface>,
   handler: Handler? = null
): CameraCaptureSession {
   val outputs = mutableListOf<OutputConfiguration>()
   val isVideoCallUseCaseSupported =
       characteristics.get(CameraCharacteristics.SCALER_AVAILABLE_STREAM_USE_CASES)
       ?.contains(CameraMetadata.SCALER_AVAILABLE_STREAM_USE_CASES_VIDEO_CALL.toLong()) ?:
false
….

 }
```

Note: CameraX can set stream use cases with inter-op APIs which are experimental

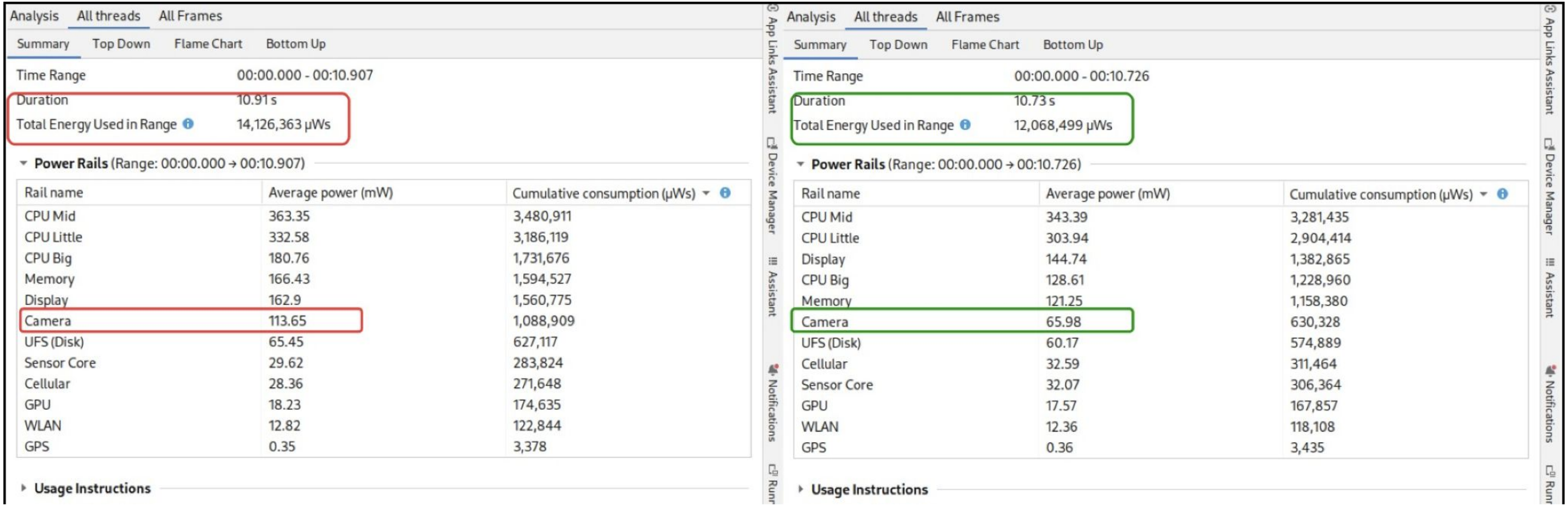
Google

Camera2 Example: VIDEO_CALL Stream Use Case

```
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
private fun createCaptureSession(
    device: CameraDevice,
    targets: List<Surface>,
    handler: Handler? = null
): CameraCaptureSession {
     ….
    for (target in targets) {
        val outputConf = OutputConfiguration(target)
        if (isVideoCallUseCaseSupported) {
            outputConf.setStreamUseCase(
                CameraMetadata.SCALER_AVAILABLE_STREAM_USE_CASES_VIDEO_CALL.toLong())
        }
        outputs.add(outputConf)
    }// Continue with session configuration
}
```

Note: CameraX can set stream use cases with inter-op APIs which are experimental

Power comparison : DEFAULT vs VIDEO_CALL



Time Range	00:00.000 - 00:10.907
Duration	10.91 s
Total Energy Used in Range	14,126,363 µWs

Power Rails (Range: 00:00.000 → 00:10.907)

Rail name	Average power (mW)	Cumulative consumption (µWs)
CPU Mid	363.35	3,480,911
CPU Little	332.58	3,186,119
CPU Big	180.76	1,731,676
Memory	166.43	1,594,527
Display	162.9	1,560,775
Camera	113.65	1,088,909
UFS (Disk)	65.45	627,117
Sensor Core	29.62	283,824
Cellular	28.36	271,648
GPU	18.23	174,635
WLAN	12.82	122,844
GPS	0.35	3,378

Usage Instructions

Time Range	00:00.000 - 00:10.726
Duration	10.73 s
Total Energy Used in Range	12,068,499 µWs

Power Rails (Range: 00:00.000 → 00:10.726)

Rail name	Average power (mW)	Cumulative consumption (µWs)
CPU Mid	343.39	3,281,435
CPU Little	303.94	2,904,414
Display	144.74	1,382,865
CPU Big	128.61	1,228,960
Memory	121.25	1,158,380
Camera	65.98	630,328
UFS (Disk)	60.17	574,889
Cellular	32.59	311,464
Sensor Core	32.07	306,364
GPU	17.57	167,857
WLAN	12.36	118,108
GPS	0.36	3,435

Usage Instructions

Android Studio Profiler on Pixel 8 using 1080p SurfaceView for preview on Camera2 Basic sample

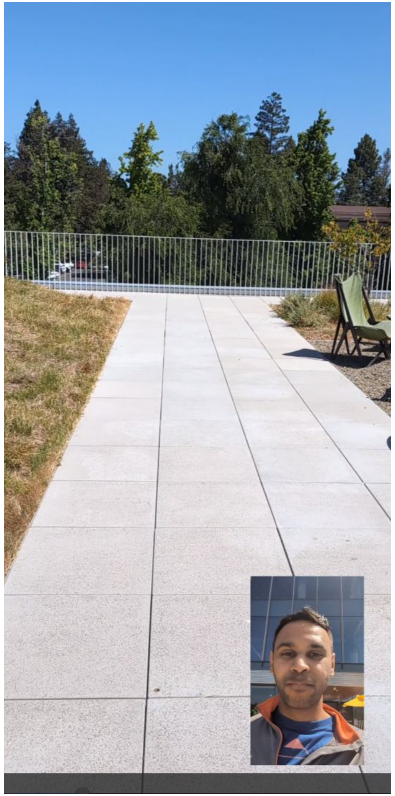
DEFAULT VIDEO_CALL

(~40% savings on camera power, 15% energy savings overall)

Concurrent Camera Streaming

Expressibility++ : Concurrent Camera Streaming

- Your app has stable video, you also have a good line of communication with the camera-subsystem to tell it what you're using your streams for
- You'd like to add a feature in your app to give it some extra points in expressibility
- Stream cameras concurrently!
  - Front + back : most popular combination



Captured on Pixel 8

Expressibility++ : Concurrent Camera Streaming

- Camera APIs by themselves don’t disallow concurrent camera streaming
  - Camera2: Just open multiple cameras and configure sessions on each camera device
- Are there guarantees though ?
  - Session creation or CaptureRequest(s) may fail
- Query through camera APIs added in API level 30

Camera2: Querying for Concurrent Camera Streaming Capabilities

- Added in API 30
  Set<Set<String>>
  CameraManager.getConcurrentCameraIds()
  - Returns combinations of camera ids that can stream concurrently
  - No restriction on camera id facing - can be FRONT + BACK, FRONT + FRONT, BACK + BACK
  - Note: All cameras must be opened before configuring sessions

Concurrent Cameras: A few of things to note

- Cameras operating concurrently may need to share the same processing pipeline, so there are limitations to the stream combinations.
- Guaranteed stream combinations in a nutshell
  - 720p PRIV + 1440p JPEG / YUV / PRIV (PRIV - Implementation defined format, typically from `SurfaceTexture` or `SurfaceView`)
  - Default CaptureRequest settings for each `CameraDevice`
  - Details on stream combinations can be found here
- For checking if non guaranteed stream combinations are supported, `CameraManager.isConcurrentSessionConfigurationSupported()` can be used.

Concurrent Camera Capability through PackageManager feature

- PackageManager.FEATURE_CONCURRENT_CAMERA
- Primary FRONT and Primary BACK cameras can stream concurrently with guaranteed stream combinations
- Primarily targeted at devices with API level < 30, for which CameraManager.getConcurrentCameraIds() wouldn't be available.
    - Apps can query PackageManager using the string "android.hardware.camera.concurrent"

Camera2: Front and Back Concurrent Cameras

```
    private fun getFrontBackConcurrentPair() : Triple<Boolean, String?, String?> {
        // Check for concurrent front back support
        var frontBackConcurrentSupported : Boolean =
           packageManager.hasSystemFeature(“android.hardware.camera.concurrent”)
        var primaryFrontId : String?  = null
        var primaryBackId : String?  = null
        if (frontBackConcurrentSupported == false) {
            return Triple(false, null, null)
        }
        ….// contd…
    }
```

Camera2: Front and Back Concurrent Cameras

```
  private fun getFrontBackConcurrentPair() : Triple<Boolean, String?, String?> {
     ….
    for (cameraId in cameraIdList) {
        val lensFacing  =
cameraManager.getCameraCharacteristics(cameraId).get(CameraCharacteristicsLENS_FACING)
        if (lensFacing == CameraMetadata.LENS_FACING_BACK) {
            primaryBackId = cameraId
        } else if (lensFacing == CameraMetadata.LENS_FACING_FRONT) {
            primaryFrontId = cameraId
        }
        if (primaryfrontId != null && primaryBackId != null) {
            return Triple(true, primaryFrontId, primaryBackId);
        }
    }
    return Triple(false, null, null)
}
```

CameraX: Front and Back Concurrent Cameras

```
var primaryFrontSelector: CameraSelector? = null
var primaryBackSelector: CameraSelector? = null
for (cameraInfoList in cameraProvider.availableConcurrentCameraInfos) {
    for (cameraInfo in cameraInfoList) {
        if (cameraInfo.lensFacing == CameraSelector.LENS_FACING_FRONT) {
            primaryFrontSelector = cameraInfo.getCameraSelector()
        } else if (cameraInfo.lensFacing == CameraSelector.LENS_FACING_BACK) {
            primaryBackSelector = cameraInfo.getCameraSelector()
        }
    }
}
if (primaryFrontSelector == null || primaryBackSelector == null) {
    return
}
```

CameraX: Front and back concurrent cameras

```
var primaryFrontSelector: CameraSelector? = null
var primaryBackSelector: CameraSelector? = null
// Set up primaryFrontSelector and primaryBackSelector
….
// Set up primary SingleCameraConfig
val previewFront = Preview.Builder().build()
previewFront.setSurfaceProvider(frontPreviewView.getSurfaceProvider())
val primaryFront = SingleCameraConfig(
    primaryFrontSelector,
    UseCaseGroup.Builder()
         .addUseCase(previewFront)
         .build(),
    lifecycleOwner
)
```

CameraX: Front and Back Concurrent Cameras

```
// Set up primary SingleCameraConfig
….
// Set up secondary SingleCameraConfig

val previewBack = Preview.Builder().build()
previewBack.setSurfaceProvider(backPreviewView.getSurfaceProvider())
val secondary = SingleCameraConfig(
   primaryBackSelector,
   UseCaseGroup.Builder()
        .addUseCase(previewBack)
        .build(),
   lifecycleOwner
)
cameraProvider.bindToLifecycle(listOf(primary, secondary))
```

Elevate your Captures

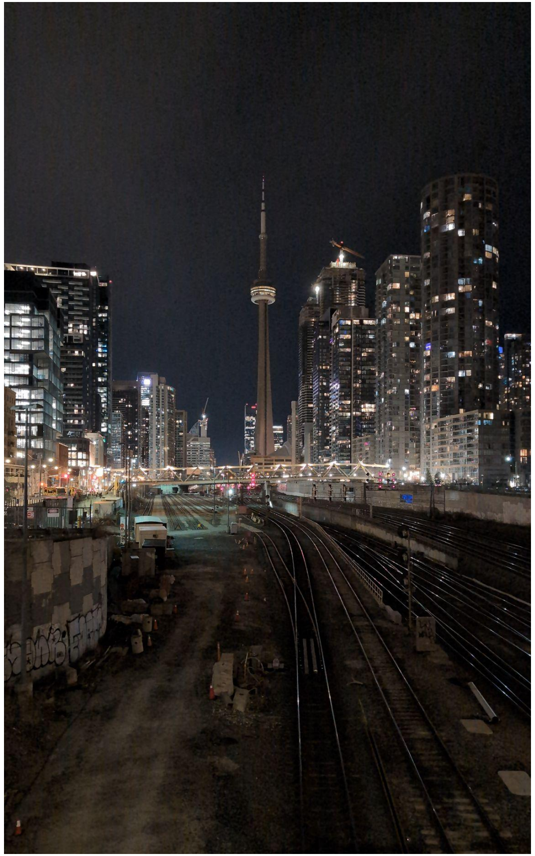
Elevate your Captures

Now our Application can

- Stabilize the preview output
- Optimize HW / SW Pipelines for scenarios
- Capture from front and rear camera at the same time

Elevate your Captures

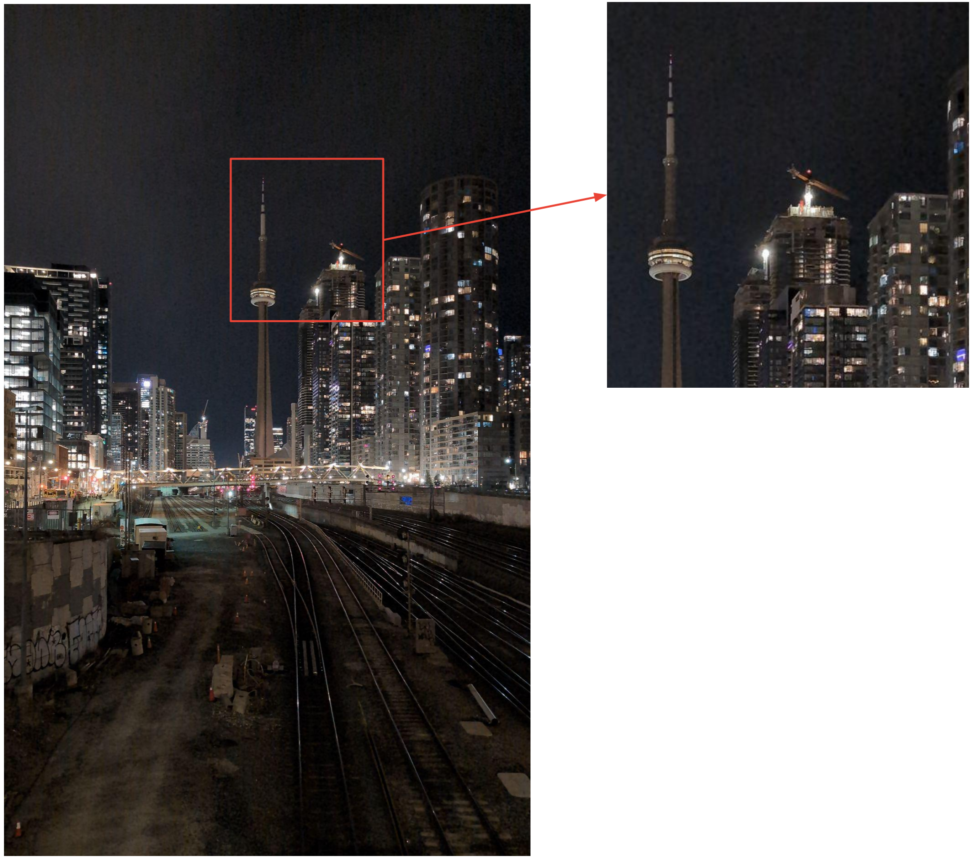
How can my app take better photos in low light?



Captured using Pixel 8

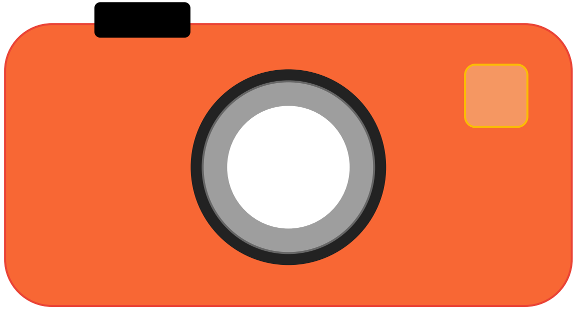
Elevate your Captures

How can my app take better photos in low light?



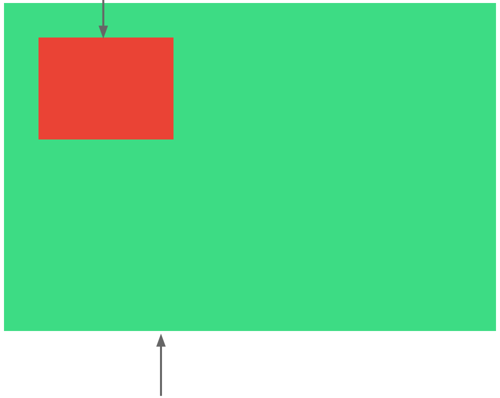
Captured using Pixel 8

Why is low light capture so hard?



DSLR

Photo Credit: Google



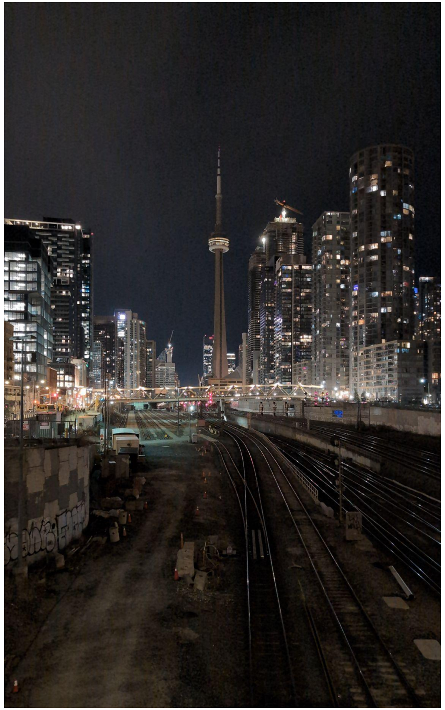
Pixel 8 Pro Sensor Size

Full Frame Sensor Size

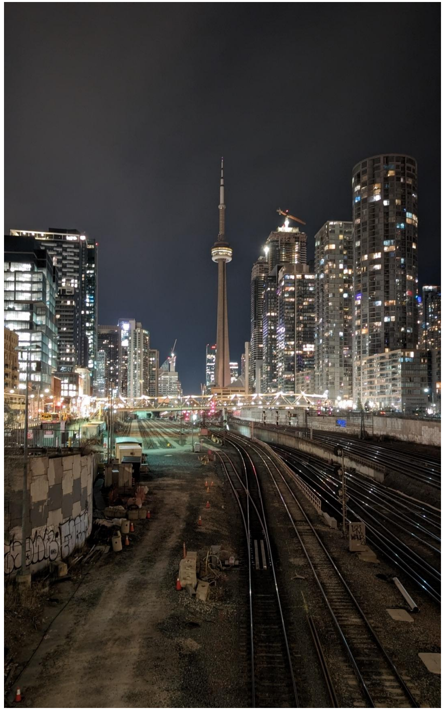
11x Larger

Google

Why is low light capture so hard?

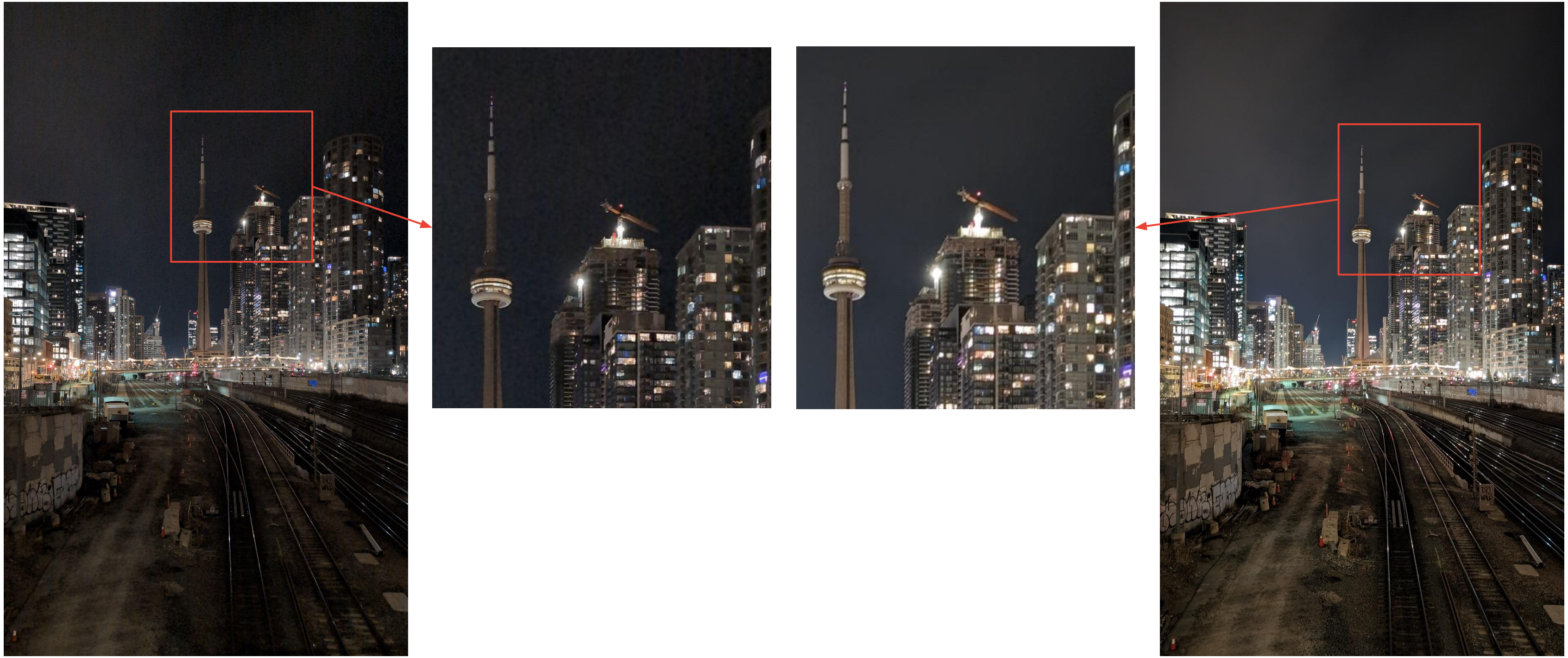


Our app
Without Night Extension



Our app
With Night Extension

Why is low light capture so hard?



Our app
Without Night Extension

Captured using Pixel 8

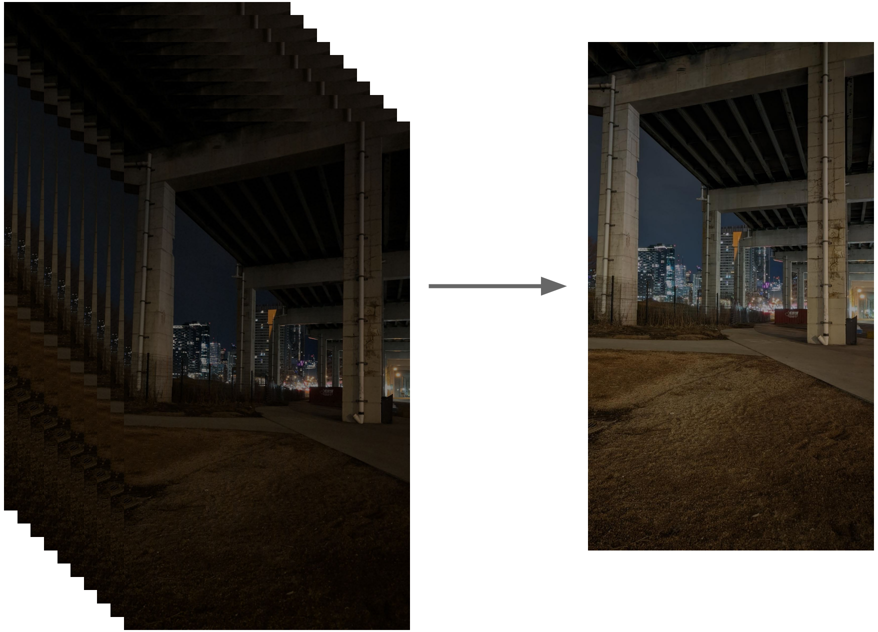
Our app
With Night Extension

How does night mode work?

Burst of RAW frames

Captured using Pixel 8

How does night mode work?



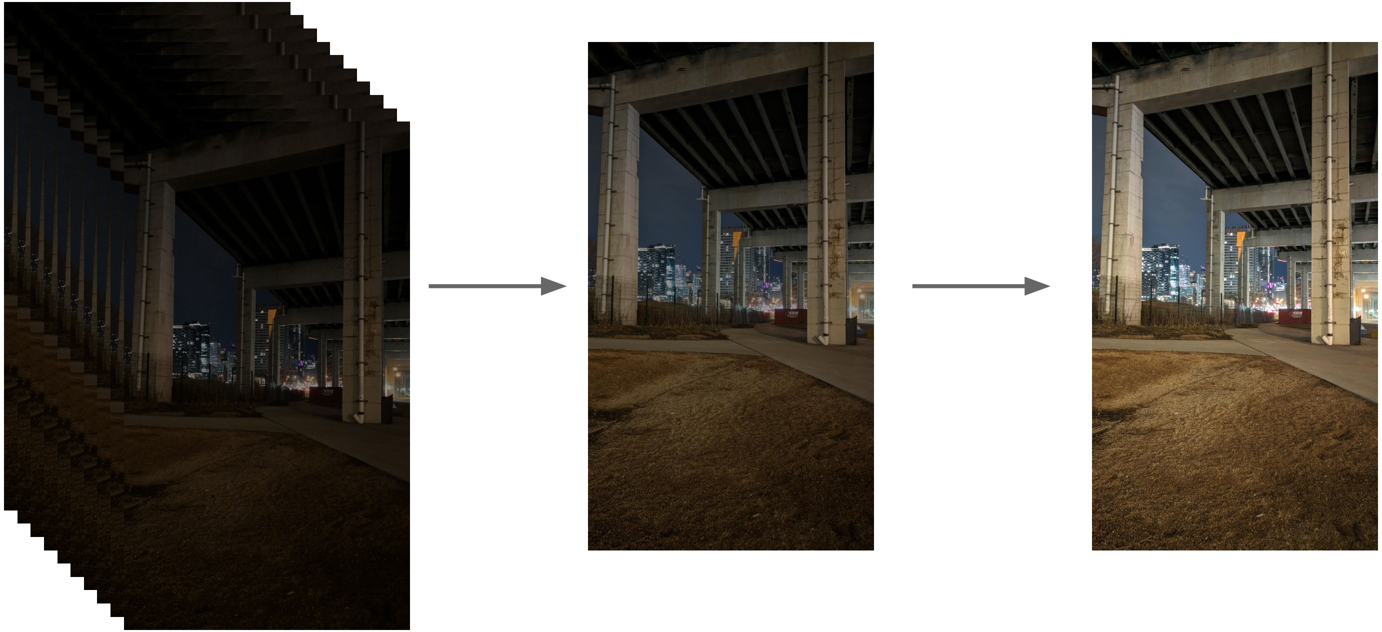
Burst of RAW frames

Merged RAW image

Captured using Pixel 8

Google

How does night mode work?



Burst of RAW frames

Merged RAW image

Final High Quality Result

Captured using Pixel 8

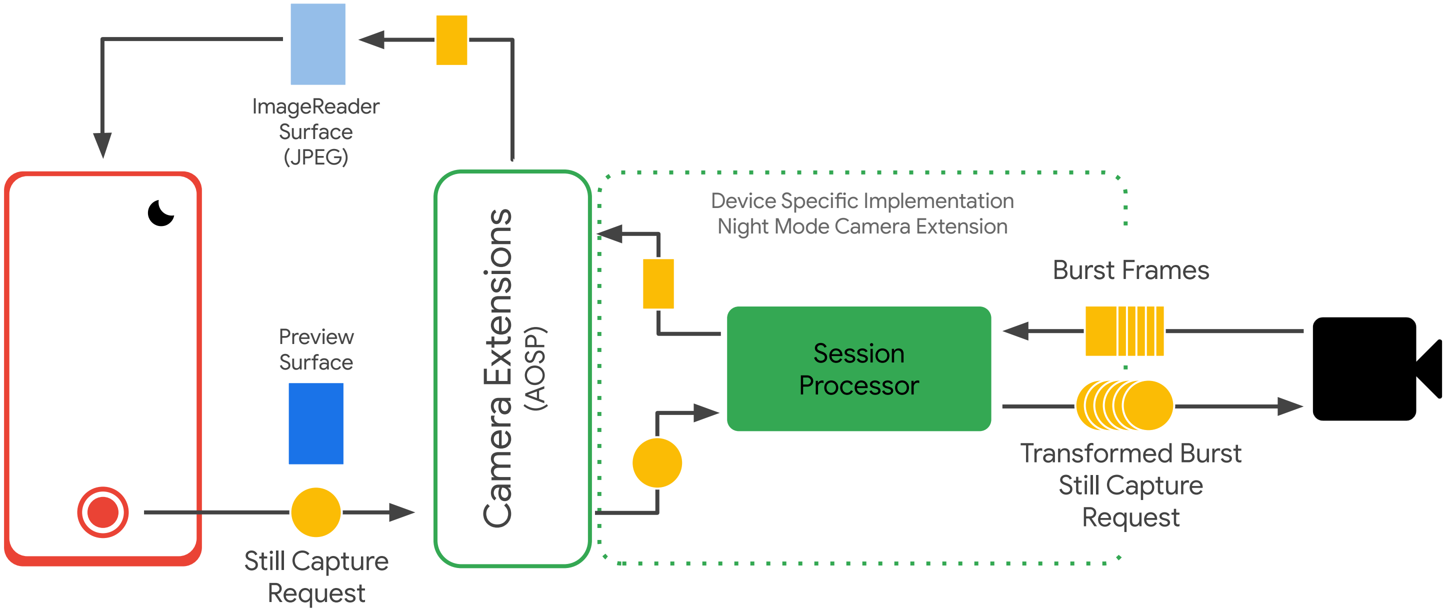
How can you add this to your app?

- Very complex logic
- Device specific == scaling challenges

Camera Extensions

- Bridge between your app and your phone's camera capabilities
- Take advantage of features like
  - Night Mode
  - Portrait
- Simple and easy to use

Camera Extensions



Google

Camera Extensions

>90

with Camera Extensions support

Adding Night Mode to your App

- Works for Camera2 and CameraX
    - Access to the exact same device implementation

Adding Night Mode using Camera2

Adding Night Mode using Camera2

```kotlin
@RequiresApi(Build.VERSION_CODES.S)
fun CameraManager.isExtensionSupported(
    cameraId: String,
    extension: Int
): Boolean =
    getCameraExtensionCharacteristics(cameraId)
        .supportedExtensions
        .contains(extension)
```

Adding Night Mode using Camera2

```
@RequiresApi(Build.VERSION_CODES.S)
private fun createExtensionCaptureSession(
    device: CameraDevice,
    configs: List<OutputConfiguration>,
    extension: Int = CameraExtensionCharacteristics.EXTENSION_NIGHT,
    executor: Executor
) {
    if (!cameraManager.isExtensionSupported(device.id, extension)) return
    // Implement callbacks
    val cb = object : CameraExtensionSession.StateCallback() {
        // Implement onConfigured & onConfigureFailed
    }
    val config = ExtensionSessionConfiguration(extension, configs, executor, cb)
    device.createExtensionSession(config)
}
```

Adding Night Mode using Camera2

```
@RequiresApi(Build.VERSION_CODES.S)
private fun createExtensionCaptureSession(
    device: CameraDevice,
    configs: List<OutputConfiguration>,
    extension: Int = CameraExtensionCharacteristics.EXTENSION_NIGHT,
    executor: Executor
) {
    if (!cameraManager.isExtensionSupported(device.id, extension)) return
    // Implement callbacks
    val cb = object : CameraExtensionSession.StateCallback() {
        // Implement onConfigured & onConfigureFailed
    }
    val config = ExtensionSessionConfiguration(extension, configs, executor, cb)
    device.createExtensionSession(config)
}
```

Adding Night Mode using Camera2

```
@RequiresApi(Build.VERSION_CODES.S)
private fun createExtensionCaptureSession(
    device: CameraDevice,
    configs: List<OutputConfiguration>,
    extension: Int = CameraExtensionCharacteristics.EXTENSION_NIGHT,
    executor: Executor
) {
    if (!cameraManager.isExtensionSupported(device.id, extension)) return
    // Implement callbacks
    val cb = object : CameraExtensionSession.StateCallback() {
        // Implement onConfigured & onConfigureFailed
    }
    val config = ExtensionSessionConfiguration(extension, configs, executor, cb)
    device.createExtensionSession(config)
}
```

Adding Night Mode using Camera2

```
@RequiresApi(Build.VERSION_CODES.S)
private fun createExtensionCaptureSession(
    device: CameraDevice,
    configs: List<OutputConfiguration>,
    extension: Int = CameraExtensionCharacteristics.EXTENSION_NIGHT,
    executor: Executor
) {
    if (!cameraManager.isExtensionSupported(device.id, extension)) return
    val cb = object : CameraExtensionSession.StateCallback() {
        override fun onConfigured(session: CameraExtensionSession) {
            startPreview(session)
        }
        ...
    }
    val config = ExtensionSessionConfiguration(extension, configs, executor, cb)
    device.createExtensionSession(config)
}
```

Adding Night Mode using Camera2

```kotlin
@RequiresApi(Build.VERSION_CODES.S)
private fun createExtensionCaptureSession(
    device: CameraDevice,
    configs: List<OutputConfiguration>,
    extension: Int = CameraExtensionCharacteristics.EXTENSION_NIGHT,
    executor: Executor
) {
    if (!cameraManager.isExtensionSupported(device.id, extension)) return
    val cb = object : CameraExtensionSession.StateCallback() {
        override fun onConfigured(session: CameraExtensionSession) {
            startPreview(session)
        }
        ...
    }
    val config = ExtensionSessionConfiguration(extension, configs, executor, cb)
    device.createExtensionSession(config)
}
```

Adding Night Mode using Camera2

```
@RequiresApi(Build.VERSION_CODES.S)
private fun capturePhoto(
    device: CameraDevice,
    extensionSession: CameraExtensionSession,
    executor: Executor
) {
    val captureBuilder =
        cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_STILL_CAPTURE)
    captureBuilder.addTarget(stillImageReader.surface)
    cameraExtensionSession.capture(
        captureBuilder.build(),
        executor,
        captureCallbacks
    )
}
```

Adding Night Mode using CameraX

```
val cameraProvider = ProcessCameraProvider.getInstance(application).await()
val useCaseGroup = UseCaseGroup.Builder() ... .build()
var cameraSelector = CameraSelector.Builder()
   .requireLensFacing(CameraSelector.LENS_FACING_BACK).build()

val extensionsManager = ExtensionsManager.getInstanceAsync(context, cameraProvider).await()
if (extensionsManager.isExtensionAvailable(cameraSelector, ExtensionMode.NIGHT)) {
   cameraSelector = extensionsManager.getExtensionEnabledCameraSelector(
       cameraSelector, ExtensionMode.NIGHT)
}

camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, useCaseGroup)
```

Adding Night Mode using CameraX

```
val cameraProvider = ProcessCameraProvider.getInstance(application).await()
val useCaseGroup = UseCaseGroup.Builder() ... .build()
var cameraSelector = CameraSelector.Builder()
    .requireLensFacing(CameraSelector.LENS_FACING_BACK).build()

val extensionsManager = ExtensionsManager.getInstanceAsync(context, cameraProvider).await()
if (extensionsManager.isExtensionAvailable(cameraSelector, ExtensionMode.NIGHT)) {
    cameraSelector = extensionsManager.getExtensionEnabledCameraSelector(
        cameraSelector, ExtensionMode.NIGHT)
}

camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, useCaseGroup)
```

Adding Night Mode using CameraX

```
val cameraProvider = ProcessCameraProvider.getInstance(application).await()
val useCaseGroup = UseCaseGroup.Builder() ... .build()
var cameraSelector = CameraSelector.Builder()
   .requireLensFacing(CameraSelector.LENS_FACING_BACK).build()

val extensionsManager = ExtensionsManager.getInstanceAsync(context, cameraProvider).await()
if (extensionsManager.isExtensionAvailable(cameraSelector, ExtensionMode.NIGHT)) {
   cameraSelector = extensionsManager.getExtensionEnabledCameraSelector(
       cameraSelector, ExtensionMode.NIGHT)
}

camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, useCaseGroup)
```

Adding Night Mode using CameraX

```
val cameraProvider = ProcessCameraProvider.getInstance(application).await()
val useCaseGroup = UseCaseGroup.Builder() ... .build()
var cameraSelector = CameraSelector.Builder()
   .requireLensFacing(CameraSelector.LENS_FACING_BACK).build()

val extensionsManager = ExtensionsManager.getInstanceAsync(context, cameraProvider).await()
if (extensionsManager.isExtensionAvailable(cameraSelector, ExtensionMode.NIGHT)) {
   cameraSelector = extensionsManager.getExtensionEnabledCameraSelector(
       cameraSelector, ExtensionMode.NIGHT)
}

camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, useCaseGroup)
```

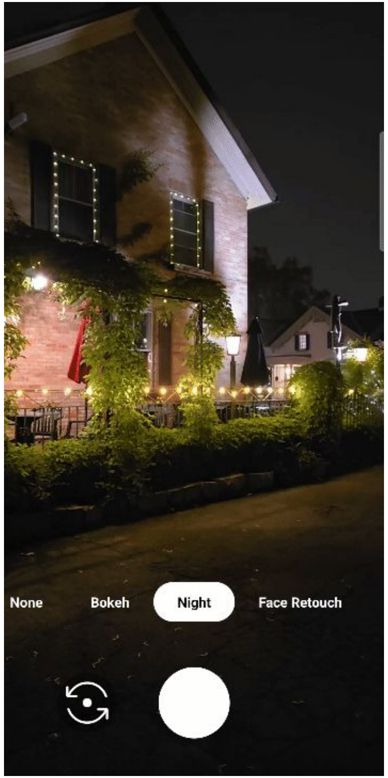
Adding Night Mode using CameraX

```
imageCapture.takePicture(
   outputFileOptions,
   Dispatchers.Main.asExecutor(),
   object : ImageCapture.OnImageSavedCallback {
       override fun onImageSaved(outputFileResults: ImageCapture.OutputFileResults) {
           imageCaptureRepository.notifyImageCreated(outputFileResults.savedUri)
       }

       override fun onError(exception: ImageCaptureException) {
           ...
       }
   })
```

UI Affordances for Latency

- Night Mode captures can take seconds
- Android 14 adds APIs to communicate the latency as part of the user journey
- Postview
- Capture Processing Progress
- Realtime Capture Latency Estimate



Google

UI Affordances for Latency with CameraX

```kotlin
val camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector)
val isPostviewSupported = ImageCapture
    .getImageCaptureCapabilities(camera.getCameraInfo())
    .isPostviewSupported()

val imageCapture = ImageCapture.Builder().setPostviewEnabled(isPostviewSupported).build()
imageCapture.takePicture(..., object : ImageCapture.OnImageSavedCallback {
    override fun onPostviewBitmapAvailable(bitmap: Bitmap) {
        showPostview(bitmap)
    }
})
```

UI Affordances for Latency with CameraX

```kotlin
val camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector)
val isPostviewSupported = ImageCapture
    .getImageCaptureCapabilities(camera.getCameraInfo())
    .isPostviewSupported()

val imageCapture = ImageCapture.Builder().setPostviewEnabled(isPostviewSupported).build()
imageCapture.takePicture(..., object : ImageCapture.OnImageSavedCallback {
    override fun onPostviewBitmapAvailable(bitmap: Bitmap) {
         showPostview(bitmap)
    }
})
```

UI Affordances for Latency with CameraX

```kotlin
val camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector)
val isPostviewSupported = ImageCapture
   .getImageCaptureCapabilities(camera.getCameraInfo())
   .isPostviewSupported()

val imageCapture = ImageCapture.Builder().setPostviewEnabled(isPostviewSupported).build()
imageCapture.takePicture(..., object : ImageCapture.OnImageSavedCallback {
   override fun onPostviewBitmapAvailable(bitmap: Bitmap) {
       showPostview(bitmap)
   }
})
```

UI Affordances for Latency with CameraX

```
val camera = cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector)
val isPostviewSupported = ImageCapture
    .getImageCaptureCapabilities(camera.getCameraInfo())
    .isPostviewSupported()

val imageCapture = ImageCapture.Builder().setPostviewEnabled(isPostviewSupported).build()
imageCapture.takePicture(..., object : ImageCapture.OnImageSavedCallback {
    override fun onPostviewBitmapAvailable(bitmap: Bitmap) {
        showPostview(bitmap)
    }
})
```

UI Affordances for Latency with CameraX

```
...
val isProcessProgressSupported = ImageCapture
   .getImageCaptureCapabilities(camera.getCameraInfo())
   .isCaptureProcessProgressSupported()
image.takePicture(... , object : ImageCapture.OnImageSavedCallback {
   ...
   override fun onCaptureProcessProgressed(progress: Int) {
       if (isProcessProgressSupported) {
           showProcessProgress(progress)
       }
   }
})
```

UI Affordances for Latency with CameraX

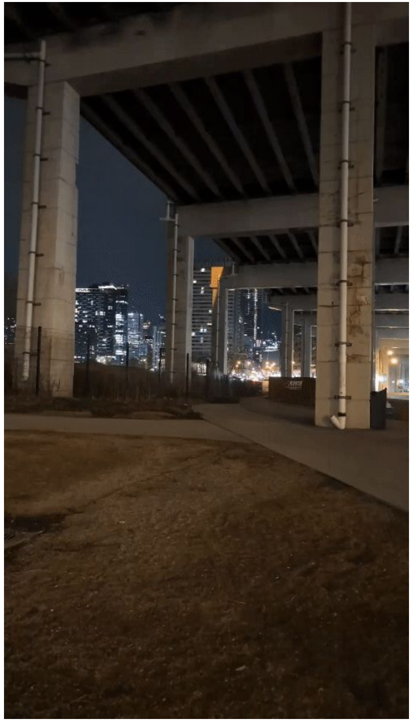
```
...
val isProcessProgressSupported = ImageCapture
    .getImageCaptureCapabilities(camera.getCameraInfo())
    .isCaptureProcessProgressSupported()
image.takePicture(... , object : ImageCapture.OnImageSavedCallback {
    ...
    override fun onCaptureProcessProgressed(progress: Int) {
        if (isProcessProgressSupported) {
            showProcessProgress(progress)
        }
    }
})
```

UI Affordances for Latency with CameraX

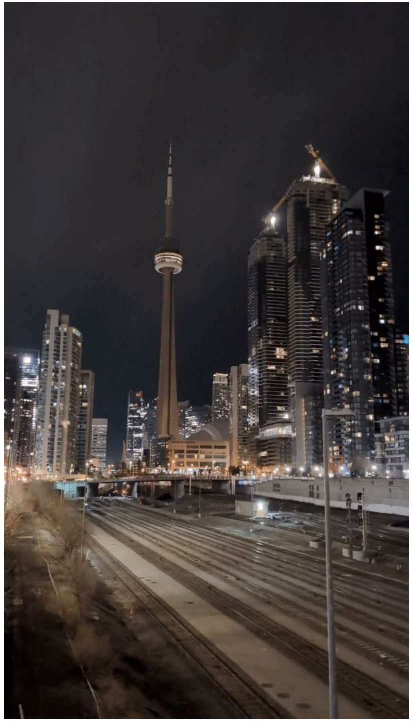
```
...
val isProcessProgressSupported = ImageCapture
   .getImageCaptureCapabilities(camera.getCameraInfo())
   .isCaptureProcessProgressSupported()
image.takePicture(... , object : ImageCapture.OnImageSavedCallback {
   ...
   override fun onCaptureProcessProgressed(progress: Int) {
       if (isProcessProgressSupported) {
           showProcessProgress(progress)
       }
   }
})
```

Low Light Boost

- New for Android 15
- Realtime boost applied to preview in low light scenes
- Automatically adjusts to different lighting conditions



Pixel 8



Pixel 8

Adding Low Light Boost in Camera2

```
fun CameraManager.isLowLightBoostAvailable(cameraId: String): Boolean =
   Build.VERSION.SDK_INT >= 35 &&
           getCameraCharacteristics(cameraId)
               .get(CONTROL_AE_AVAILABLE_MODES)
               ?.contains(CONTROL_AE_MODE_ON_LOW_LIGHT_BOOST_BRIGHTNESS_PRIORITY)
           ?: false
```

Adding Low Light Boost in Camera2

```
val request =
    cameraSession.device.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW).apply {
        if (cameraManager.isLowLightBoostAvailable(cameraId)) {
            set(
                CONTROL_AE_MODE, CONTROL_AE_MODE_ON_LOW_LIGHT_BOOST_BRIGHTNESS_PRIORITY
            )
        }
    }.build()
```

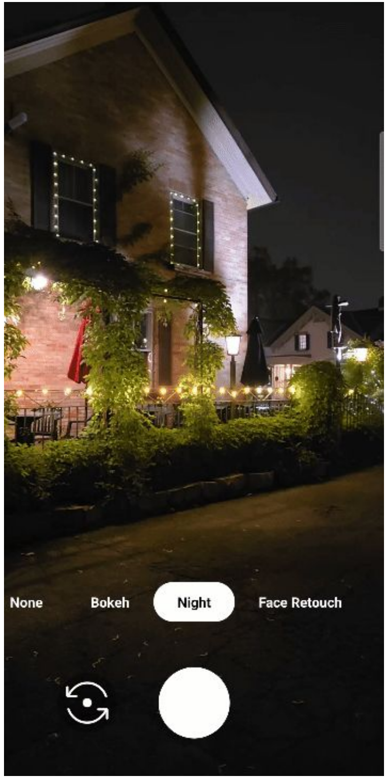
Adding Low Light Boost in Camera2

```
cameraSession.setRepeatingRequest(request, object : CameraCaptureSession.CaptureCallback() {
    override fun onCaptureCompleted(
        session: CameraCaptureSession, request: CaptureRequest, result: TotalCaptureResult
    ) {
        if (result.get(CONTROL_LOW_LIGHT_BOOST_STATE) == CONTROL_LOW_LIGHT_BOOST_STATE_ACTIVE) {
            showNightMode()
        } else {
            hideNightMode()
        }
    }
}, handler)
```

Elevate your Captures

Now our Application can

- Capture stunning photos in low light
- Communicate the latency as part of the user journey
- Automatically adapt the preview brightness to low light conditions

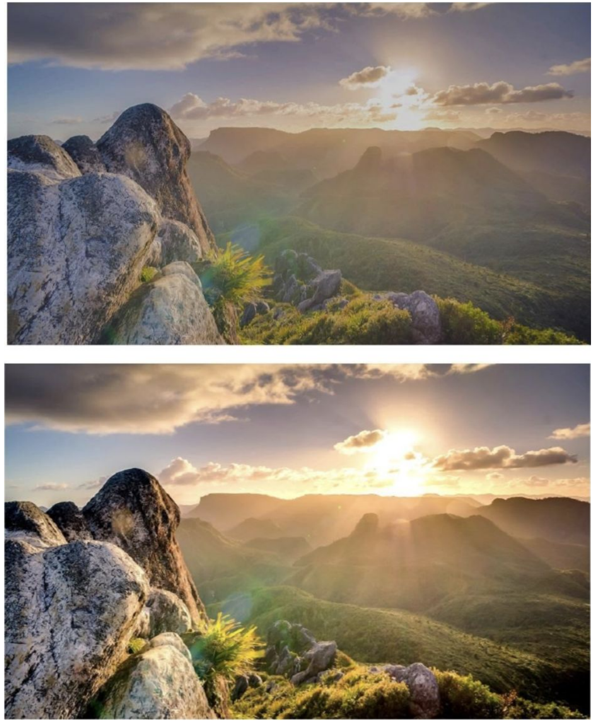


HDR Video and UltraHDR

HDR Video (Android 13)

- HDR Video enhances the in-app experience for video capture, playback, edit and share by enabling vibrant color and great contrast.
- HDR Video captures and display video in 10-bit with different flavors (**HLG10**, HDR10, HDR10+, Dolby Vision).

Capture Playback Edit Share

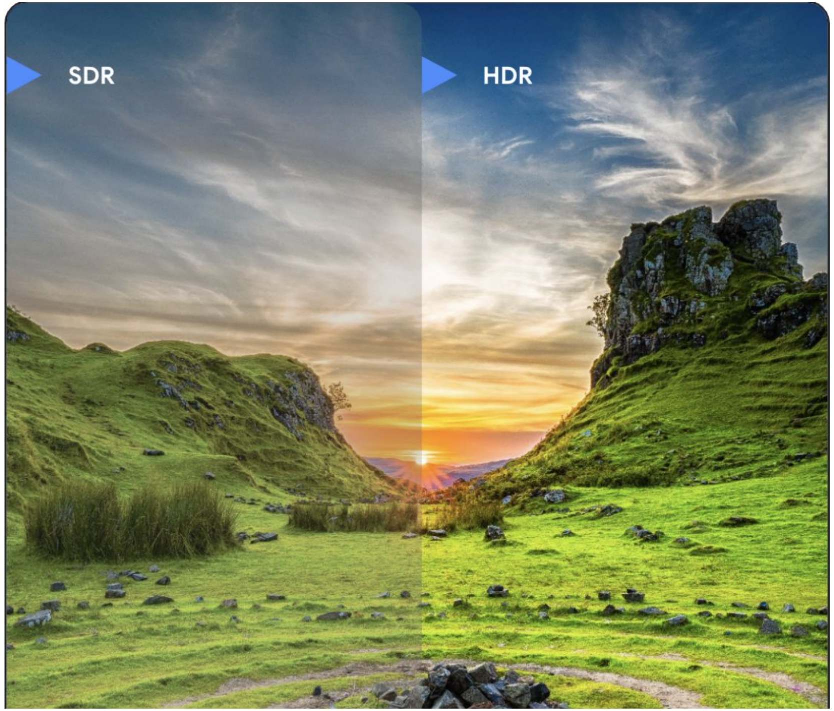


Credit: Google

UltraHDR (Android 14)

- UltraHDR is a new technology that helps capture, render, edit and share image in HDR on Android supported devices.
- UltraHDR delivers image in more detailed highlights and shadows.
- UltraHDR stores image in **JPEG/R** and is backwards compatible with JPEG.

Capture Render Edit Share



Credit: Google

Google

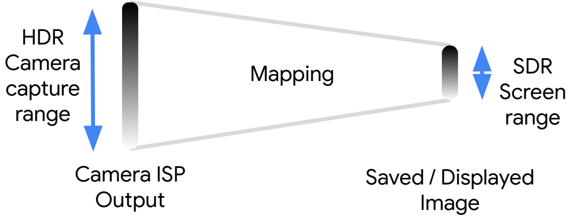
SDR vs HDR Display

Before: HDR Images mapped to SDR display range

Camera ISP Output

Displayed/ saved to file

Exposure bracketing to capture higher dynamic range but mapped for SDR display and saving to file.

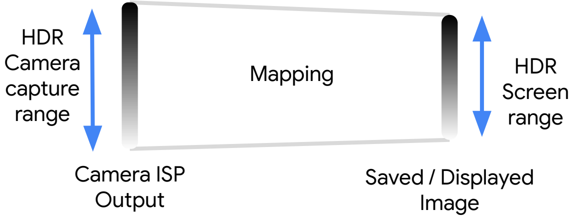


14: HDR Image range preserved for HDR displays

Camera ISP Output

Displayed/ saved to file

Preserving the dynamic range of images to display on HDR capable displays in 10-bit.



Credit: Google

Illustration

How to build HDR Video capture

HDR Video Capture in Camera2 (Capability)

```kotlin
// Check if Camera has 10-Bit output capabilities
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun CameraManager.isTenBitSupported(cameraId: String): Boolean =
    getCameraCharacteristics(cameraId)
        .get(CameraCharacteristics.REQUEST_AVAILABLE_CAPABILITIES)
        ?.contains(
            CameraMetadata.REQUEST_AVAILABLE_CAPABILITIES_DYNAMIC_RANGE_TEN_BIT
        )
        ?: false
```

HDR Video Capture in Camera2 (Capability)

```kotlin
// Check if Camera is supporting HLG10 video format
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun CameraManager.isHLGSupported(cameraId: String): Boolean =
    getCameraCharacteristics(cameraId)
       .get(CameraCharacteristics.REQUEST_AVAILABLE_DYNAMIC_RANGE_PROFILES)
       ?.supportedProfiles
       ?.contains(DynamicRangeProfiles.HLG10) ?: false
```

HDR Video Capture in Camera2 (Capability)

```
// Check if Camera is supporting HLG10 video format
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun CameraManager.isHDRVideoSupported(cameraId: String): Boolean =
    getCameraCharacteristics(cameraId)
       .get(CameraCharacteristics.REQUEST_AVAILABLE_DYNAMIC_RANGE_PROFILES)
       ?.supportedProfiles
       ?.contains(DynamicRangeProfiles.HLG10) ?: false
```

- HLG10 is the **minimum** required profile for **capture and playback** to provide better consistency of experiences across mobile devices

HDR Video Capture in Camera2 (Configure)

```kotlin
// Create a capture session with configuration for HLG10
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun configureSession(device: CameraDevice, targets: List<Surface>,
    exe: Executor, cb: CameraCaptureSession.StateCallback
) {
    val configs = targets.map { surface ->
        val config = OutputConfiguration(surface)
        config.dynamicRangeProfile = DynamicRangeProfiles.HLG10
        config
    }
    val session = SessionConfiguration(SessionConfiguration.SESSION_REGULAR,
        configs, exe, cb)
    device.createCaptureSession(session)
}
```

HDR Video Capture in Camera2 (Configure)

```
// Create a capture session with configuration for HLG10
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun configureSession(device: CameraDevice, targets: List<Surface>,
    exe: Executor, cb: CameraCaptureSession.StateCallback
) {
    val configs = targets.map { surface ->
       val config = OutputConfiguration(surface)
       config.dynamicRangeProfile = DynamicRangeProfiles.HLG10
       config
    }
    val session = SessionConfiguration(SessionConfiguration.SESSION_REGULAR,
       configs, exe, cb)
    device.createCaptureSession(session)
}
```

Google

HDR Video Capture in Camera2 (Configure)

```
// Create video encoder supporting HLG10
@RequiresApi(Build.VERSION_CODES.N)
fun configureEncoder(surface: Surface, w:Int, h:Int) {
    val format = MediaFormat.createVideoFormat(MediaFormat.MIMETYPE_VIDEO_HEVC, w, h)
    // Other Setup
    // ...

    // Set media format properties
    format.setInteger(...)
}
```

HDR Video Capture in Camera2 (Configure)

```
/// Color Format
(MediaFormat.KEY_COLOR_FORMAT,MediaCodecInfo.CodecCapabilities.COLOR_FormatSurface)

/// HEVC (H.265)
(MediaFormat.KEY_PROFILE, MediaCodecInfo.CodecProfileLevel.HEVCProfileMain10)

/// HLG Color Transfer
(MediaFormat.KEY_COLOR_TRANSFER, MediaFormat.COLOR_TRANSFER_HLG)

/// BT2020 Color Standard
(MediaFormat.KEY_COLOR_STANDARD, MediaFormat.COLOR_STANDARD_BT2020)
```

HDR Video Capture in CameraX (Capability)

```
// Check if Camera has 10-Bit output capabilities
@RequiresApi(Build.VERSION_CODES.TIRAMISU)
fun isTenBitSupported(cameraInfo: CameraInfo): Boolean =
    Recorder.getVideoCapabilities(cameraInfo)
       .supportedDynamicRanges
       .contains(DynamicRange.HLG_10_BIT)
```

HDR Video Capture in CameraX (Configure)

```
// Bind use cases to start camera
val preview = Preview.Builder()
    .build()

val recorder = Recorder.Builder()
    .setQualitySelector(QualitySelector.from(Quality.HD))
    .build()
val videoCapture = VideoCapture.Builder<Recorder>(recorder)
    .setMirrorMode(MirrorMode.MIRROR_MODE_ON_FRONT_ONLY)
    .setDynamicRange(DynamicRange.HLG_10_BIT)
    .build()

...
val camera = cameraProvider.bindToLifecycle(lifecycleOwner,
    CameraSelector.LENS_FACING_FRONT,
    preview,
    videoCapture)
```

Rendering in HDR UI

By default, Activities render UI in SDR. You can opt-in to using HDR UI for an Activity by doing one of the following:

Manifest entry:

In your AndroidManifest.xml, specify `android:colorMode="hdr"`on the Activity

At runtime:

In the onCreate() lifecycle method of the Activity, set the color mode with:

```
window.colorMode = ActivityInfo.COLOR_MODE_HDR
```

SurfaceView can support HDR video playback, TextureView will tonemap HDR to SDR.

How to build UltraHDR capture

UltraHDR Capture in Camera2 (Capability)

```kotlin
// Check if Camera has 10-Bit output + JPEG_R capabilities
@RequiresApi(Build.VERSION_CODES.UPSIDE_DOWN_CAKE)
fun CameraManager.isUltraHDRSupported(cameraId: String): Boolean {
    val isTenBitSupported = getCameraCharacteristics(cameraId)
        .get(CameraCharacteristics.REQUEST_AVAILABLE_CAPABILITIES)
        ?.contains(CameraMetadata.REQUEST_AVAILABLE_CAPABILITIES_DYNAMIC_RANGE_TEN_BIT)
        ?: false
    val formats = getCameraCharacteristics(cameraId)
        .get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP)?.outputFormats
    val canEncodeUltraHDR = formats?.contains(ImageFormat.JPEG_R) ?: false

    return isTenBitSupported && canEncodeUltraHDR
}
```

UltraHDR Capture in Camera2 (Configure)

```kotlin
// Set up image reader with JPEG_R format for capturing
@RequiresApi(Build.VERSION_CODES.UPSIDE_DOWN_CAKE)
fun setUpImageReader() {
    val pixelFormat = ImageFormat.JPEG_R
    val configMap = characteristics.get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP)
    configMap?.let { config ->
        config.getOutputSizes(pixelFormat).maxByOrNull { it.height * it.width }
            ?.let { size ->
                imageReader = ImageReader.newInstance(
                    size.width, size.height, pixelFormat, IMAGE_BUFFER_SIZE,
                )
            }
    }
}
```

UltraHDR Capture in Camera2 (Configure)

```kotlin
// Create a capture session with configuration
fun configureSession(device: CameraDevice) {
    ...
    val previewConfiguration = OutputConfiguration(binding.viewfinder.holder.surface)
    previewConfiguration.dynamicRangeProfile = DynamicRangeProfiles.HLG10
    val imageCaptureConfiguration = OutputConfiguration(imageReader.surface)
    val targets = listOf(
        previewConfiguration,
        imageCaptureConfiguration,
    )
    session = device.createCaptureSession(targets)
    ...
}
```

UltraHDR Capture in Camera2 (Configure)

```
// Take photo
fun takePhoto() {
    imageReader.setOnImageAvailableListener(
        {
            val image = it.acquireLatestImage()
            val buffer = image.planes[0].buffer
            // save to disk or convert to Bitmap for display
            ...
        },
        handler,
    )
}
```

UltraHDR Capture in CameraX (Capability)

```kotlin
// Check if Camera has 10-Bit output + JPEG_R capabilities
@RequiresApi(Build.VERSION_CODES.UPSIDE_DOWN_CAKE)
fun isUltraHDRSupported(cameraInfo: CameraInfo): Boolean =
    ImageCapture.getImageCaptureCapabilities(cameraInfo)
        .getSupportedOutputFormats()
        .contains(ImageCapture.OUTPUT_FORMAT_JPEG_ULTRA_HDR)
}
```

UltraHDR Capture in CameraX (Configure)

```
// Bind use cases to start camera
val preview = Preview.Builder()
    .build()
val imageCapture = ImageCapture
    .setCaptureMode(ImageCapture.CAPTURE_MODE_MAXIMIZE_QUALITY)
    .setOutputFormat(ImageCapture.OUTPUT_FORMAT_JPEG_ULTRA_HDR)
    .build()
...
val camera = cameraProvider.bindToLifecycle(lifecycleOwner,
    CameraSelector.LENS_FACING_FRONT,
    preview,
    imageCapture)
```

Rendering in HDR UI

By default, Activities render UI in SDR. You can opt-in to using HDR UI for an Activity by doing one of the following:

Manifest entry:

In your AndroidManifest.xml, specify `android:colorMode="hdr"`on the Activity

At runtime:

In the onCreate() lifecycle method of the Activity, set the color mode with:

```
window.colorMode = ActivityInfo.COLOR_MODE_HDR
```

ImageView can support UltraHDR image rendering alongside SDR assets.

You can also use standard Bitmap APIs to manipulate images in UltraHDR.

Closing Thoughts

- Camera2 and CameraX APIs provide many capabilities to enhance your camera app
- We want to hear your feedback!
  Constantly looking for ways to innovate, improve developer experience, and user experience
- Join our developer forum for feedback and questions
  - https://groups.google.com/a/android.com/g/camerax-developers

THANK YOU

Reference

- https://developer.android.com/media/camera/camera2
- https://source.android.com/docs/core/camera/concurrent-streaming
- https://developer.android.com/media/camera/camera2/hdr-video-capture
- https://developer.android.com/media/grow/hdr-playback
- https://developer.android.com/media/grow/ultra-hdr
- https://developer.android.com/media/platform/hdr-image-format
- https://developer.android.com/about/versions/15/features/low-light-boost
- https://developer.android.com/media/camera/camera-extensions
- https://github.com/android/camera-samples

Supplemental Slides

Which API do I use : Camera2 or CameraX ?

- In general, we **recommend using CameraX**, it'll simplify your camera development quite a bit
  - If you're building a camera app for the first time, use CameraX
- However, if your app needs to support complex use cases - for example: you want to process your own RAW captures or deal with high speed capture sessions, use camera2
- There's also the facility of inter-op between CameraX and camera2 for some use cases.
- Camera1 API is deprecated - **strongly recommend migrating off** it

Which API do I use : Camera2 or CameraX ?

Use case	CameraX	Camera2	Camera1
Ease of use	✅	❌	✅
Managed camera lifecycles	✅	❌	❌
Automatic Handling of different device form factors	✅	❌	
Actively maintained	✅	✅	❌
RAW and high speed capture session support	❌	✅	❌

Concurrent cameras: A few things to note

- Each camera streaming

Target1		Target2	
Format	Size	Format	Size
YUV	s1440p		
PRIV	s1440p		
JPEG	s1440p		
YUV/ PRIV	s720p	JPEG	s1440p
YUV / PRIV	s720p	YUV / PRIV	s1440p

[1] - s720p refers to the camera device's maximum resolution for that format from StreamConfigurationMap#getOutputSizes or 720p(1280X720) whichever is lower

[2] - s1440p refers to the camera device's maximum resolution for that format from StreamConfigurationMap#getOutputSizes or 1440p(1920X1440) whichever is lower.