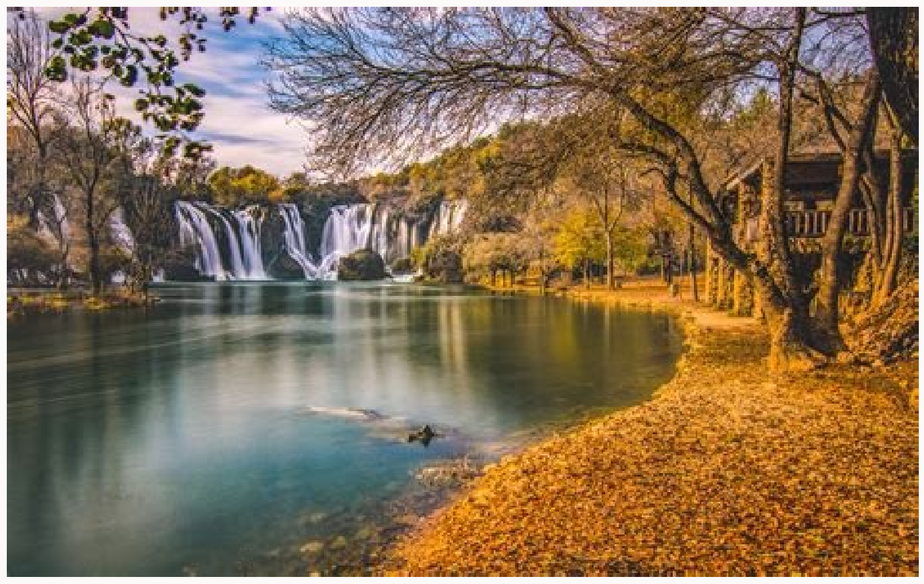
Unity best resolution for android

Continue



Unity camera size in pixels. How to set game resolution in unity. Unity how to get screen resolution.

Explore Intel Arc Discrete Graphics, create and stream with new hardware, software and services. See Xe Super Sampling in action This new AI-based API uses deep learning to synthesize high-quality images. Forums > Welcome to the Unity Forums > Beta and Experimental Features > Archived Beta > 2019.3 Beta > 2019.3 Beta Discussion Started Removed June 3, 2019 Android Vitals is Google's initiative to improve the technical quality of Google Play. application for Android devices. When a signed-in user launches your app, their Android device records quality aspects, including stability, performance, battery usage, and permission denials. This data is collected by Google Play and can be accessed in two ways: through the Google Play Console in the Android Vitals panel and through the Google Play Developer Reports API. Developers must monitor all critical elements to ensure they do not negatively impact the user experience. Specifically, developers should prioritize two key elements: user-perceived failure rate and user-perceived ANR. Highlighting and Poor Performance Your app's best features affect your app's visibility on Google Play. Each critical item has a general misbehavior threshold and a device-specific misbehavior threshold. These are documented below and also displayed in Android Vitals. Frequently Asked Questions What are the most important elements? Basic vitals are the most important indicators of Android vitals and affect your app's visibility on Google Play. The main ones include user-perceived failure rate and user-perceived ANR. What are the boundaries for bad behavior? Each critical item has two misbehavior thresholds: an overall misbehavior threshold that considers all sessions on all devices, and a misbehavior threshold that is assessed per device (phone only). Thresholds are displayed in Android Vitals. GOOD BEHAVIOR THRESHOLD To maximize your title's visibility on Google Play, follow these limits. Overall (avgdevices) For each phone model Bounce rate according to users 1.09% 8% ANR according to users 0.47% 8% How do vital signs affect the visibility of my game on Google Play? If your app or game exceeds the overall low performance threshold for any critical element, Play may limit your game's visibility to users on all device models. If your app or game exceeds the device failure threshold for a critical component for a specific device model, Play may limit the visibility of your app or game to users of that device model. Play may also display a warning on the store's listing page to let users know that your app may not work properly on their devices. Can there be bad behavior both for each device and in general? Or one and not the other? What should I do if this is the case? Yes, all combinations are possible. To improve overall quality, try to address crashes and ANR clusters that typically affect most users. To improve the quality of each device, it is necessary to eliminate the largest sets of failures and ANRs on it. If you have both, it makes sense to start with hard crashes and ANR clusters. This will likely also improve the quality of key models of individual devices. I need help with my technical problems. Where to begin? We've compiled a variety of resources to help you get started diagnosing and fixing technical issues with your app or game. Key Features: User Perceived ANR User Perceived Crash Rate All other relevant: Excessive Activations, Partial Activation Block, Excessive Wi-Fi Background Scanning Excessive Background Network Usage Application Launch Time Slow rendering based on last 28 days of data. Android Vitals will alert you to bad behavior for 28 days. Test the user interface regularly, or consider using the Reporting API to incorporate data directly into your workflows. [Coming soon] Set up email alerts in the Play Console for inappropriate behavior. [Coming soon] Android VitalsReport emerging issues, defined as devices that do not reach the misbehavior threshold for each device within 7 days. This gives you up to 21 days to prevent the problem. I have a lot of misbehaving devices. How do I understand the list? Sometimes certain aspects of your device's hardware or software can affect your spending quotas. To help you identify and fix these issues, the new Android Vitals feature alerts you when we detect potential correlations between high-level issues and key device attributes such as RAM, Android version, and system-on-chip, among others. You can also explore device reach and attribution yourself in the Play Console. With Android Vitals, you can also access consolidated device information with one click, including install base, revenue, ratings, and reviews. This information is displayed in the sidebar, so you don't have to leave the current page. How long will it take for me to stop seeing notifications when troubleshooting my device? Play evaluates your key metrics daily based on a 28-day moving average. Once the moving average drops below the thresholds again, Android Vitals will stop displaying alerts. Store list warnings can be removed even earlier if the game's algorithms detect that your technical quality is already moving in the right direction. What if I can't or don't want to solve the problem? Make sure you consider both the cost and the potential for lasting negative effects when deciding on the next course of action. Bad behavior affects existing users and reduces your ability to attract potential users. If it is not possible to fix the bad behavior on every device, you should rethink the device targeting and exclusion logic. Why doesn't the number and frequency of issues with Android Vitals match the number and frequency of issues I see with my own or third-party solutions? Android Vitals is a trusted source of information for Google Play to assess technical quality. The number and frequency of issues with Android Vitals may not match data from other sources.for several reasons: Android Vitals information comes from the Android platform and captures some events that the SDK does not see, including: Startup failures that occur before the SDK initializes ANR errors, before Android 12 Vitals issues on Android only occur when reporting certified devices and applications installed from Google Play. Other data sources may not apply these restrictions. Android Vitals only collects data from users who have agreed to share usage and diagnostic information. Other tools may not require your consent. Even if they do, it's unlikely that they're collecting data from the exact same users who chose Android. To protect users' privacy, we only display data on the dashboard when we have managed to collect enough data to create anonymous reports. Emission values can be calculated in several ways. The Android Vitals issue rate is the number of issues (crashes, ANR errors, etc.) per active user per day. Crashlytics counts problems per application session. For example, if a user played a game 3 times in one day and it crashed, Android Vitals shows a 100% crash rate, while Crashlytics shows a 33% crash rate. For more information about how data is collected, see Play Console Help. In addition to the daily free educational eBook, you have access to more than 30 premium titles that we have carefully selected for quality in various technology areas. You can access it online forever with a free account. You configure each CameraX use case to control different aspects of the use case's operations. For example, when using image capture, you can set the target aspect ratio and flash mode. The following code shows an example: val imageCapture = ImageCapture.Builder() .setFlashMode(...) .setTargetAspectRatio(...) .build() ImageCapture imageCapture = new ImageCapture.Builder() .setFlashMode(...) . setTargetAspectRatio(...) .build(); In addition to configuration options, they provide some API use casesChange the settings after creating the use case. For use case-specific configuration, see Implementation Preview, Image Analysis, and Image Capture. CameraXConfig For simplicity, CameraX has default configurations such as internal executors and handlers that are suitable for most use cases. However, if your application has special needs or you want to customize these configurations, CameraXConfig has an interface for that. Using CameraXConfig, an application can do the following: Usage Pattern This procedure describes how to use CameraXConfig: For example, in the following code example, CameraX logging is limited to error messages: class CameraApplication : Application(), CameraXConfig.Provider { ignore fun getCameraXConfig () : CameraXConfig { return CameraXConfig.Builder.fromConfig(Camera2Config.defaultConfig()) .setMinimumLoggingLevel(Log.ERROR).build() } } Store a local copy of the CameraXConfig object if the application needs to know the CameraX configuration. it . Camera Limiter During the first call to ProcessCameraProvider.getInstance(), CameraX enumerates and queries the characteristics of the cameras available on the device. Because CameraX needs to communicate with hardware components, this process can be time-consuming for any camera, especially low-end devices. If your app only uses certain device cameras, such as B. as the default front-facing camera, you can set CameraX to ignore other cameras, which can reduce the delay in launching the cameras your app uses. If the CameraSelector passed to CameraXConfig.Builder.setAvailableCamerasLimiter() filters a camera, CameraX acts as if that camera does not exist. For example, the following code restricts the application to use only the device's rear camera by default: class MainApplication : Application(), CameraXConfig.Provider { override fun getCameraXConfig(): CameraXConfig { return.setAvailableCamerasLimiter(CameraSelector.DEFAULT_BACK_CAMERA).build() } } Threads Many of the platform APIs on which CameraX is built require hardware interprocess communication (IPC) to be locked, which can sometimes take hundreds of milliseconds. For this reason, CameraX calls these APIs only from background threads, ensuring that the main thread is not blocked and the UI remains smooth. CameraX manages these background threads internally, so this behavior is transparent. However, some applications require strict thread control. CameraXConfig allows the application to set background threads used by CameraXConfig.Builder.setCameraExecutor() and CameraXConfig.Builder.setSchedulerHandler(). Note: If you provide your own launcher or schedule handler, we recommend using one that does not run code on the main thread. Camera Executor The Camera Executor is used for all calls to and from the internal camera platform API. CameraX assigns and manages an internal trigger to perform these tasks. However, if your application requires tighter thread control, use CameraXConfig.Builder.setCameraExecutor(). Schedule handler The schedule handler is used to schedule internal tasks at fixed intervals, such as B. Attempt to reopen the camera if it is unavailable. This tool does not run any tasks and only sends them to the camera launcher. It is also sometimes used on older API platforms that require callback support. In these cases, downloads will still only be sent to camera manufacturers. CameraX allocates and manages an internal HandlerThread thread to perform these tasks, but it can be overridden with CameraXConfig.Builder.setSchedulerHandler(). Logging CameraX logging allows applications to filter Logcat messages, which is a best practice to avoid verbose messages in production code. CameraX supports four levels of logging, from most detailed to most detailedLog.DEBUG (default) Log.INFO Log.WARN Log.ERROR For a detailed description of these log levels, see the Android log documentation. Use CameraXConfig.Builder.setMinimumLoggingLevel(int) to set the appropriate logging level for your application. Automatic selection CameraX automatically provides features specific to the device the application is running on. For example, CameraX will automatically determine the best resolution to use if you do not specify a resolution or if the specified resolution is not supported. All of this is handled by a library which saves you writing device specific code. The goal of CameraX is to successfully initiate a camera session. This means that CameraX makes compromises in terms of resolution and aspect ratio depending on the capabilities of the device. Compromise can occur for the following reasons: The device does not support the requested permissions. The device has compatibility issues, such as older devices that require certain permissions to function properly. On some devices, some formats are only available in certain aspect ratios. The device prefers "nearest mod16" for JPEG or video encoding. See SCALER_STREAM_CONFIGURATION_MAP for more information. Although CameraX creates and manages the session, you should always check the returned image dimensions in the use case output in your code and adjust them accordingly. Rotation By default, the rotation of the camera is set according to the default view rotation when creating the use case. In this case, CameraX produces output by default that allows the app to easily match what you expect in the preview. You can change the rotation to a custom value to support multiple display devices by passing in the current display orientation when setting use case objects or dynamically after creating them. Your application can set the rotation target through configuration options. Then it can update the rotation settings using methods from the use case API (such aseven if the life cycle works. You can use this when the app is locked in portrait mode - so there's no configuration change when rotated - but your photo or analytics use case needs to consider the device's current rotation. For example, rotation detection may be required to ensure that faces are properly oriented for face detection, or photos may be oriented horizontally or vertically. Captured image data can be saved without rotation information. The Exif data includes rotation information so gallery apps can display the image in the correct orientation when saved. To display the preview data in the correct orientation, you can use the output metadata of the Preview.PreviewOutput() function to generate a transformation. The following code example shows how to set the rotation for the orientation event: : Int) { // Tracks the orientation values to determine the target rotation value valrotation : Int = when (orientation) { in 45..134 -> Surface .ROTATION_270 in 135. 224 -> Area.ROTATION_180 in 225..314 -> Area.ROTATION_90 otherwise -> Area.ROTATION_0 } imageCapture.targetRotation = rotation } } targetEventListener.enable() } void @O ImageCapture imageCapture = new ImageCapture .Builder().build( ); OrientationEventListener OrientationEventListener = new OrientationEventListener((Context)this) { @Override public void onOrientationChanged(int Orientation) { int rotation; // Tracks the orientation values to determine the rotation value of the target if (orientation >= 45 && orientation < 135) { rotation = surface.ROTATION_270; } else if (orientation >= 135 && orientation < 225) { rotation = surface.ROTATION_180; } else if (orientation >= 225 && orientation < 315) { rotation = surface.ROTATION_90; } else { rotation = surface.ROT_0; }} }; orientationEventListener.enable(); } Depending on the specified rotation, each use case will either directly rotate the image data or provide rotation metadata to the recipients of the unrotated image data. Preview: Metadata output is provided so that the target resolution rotation is known via Preview.getTargetRotation(). Image Analysis: Metadata output is provided such that the coordinates of the image buffer are known relative to the display coordinates. ImageCapture: Exif image metadata, buffer, or both buffer and metadata are changed to indicate rotation settings. The changed value depends on the HAL implementation. Clipping rectangle By default, the clipping rectangle is a full buffer rectangle. You can customize it with ViewPort and UseCaseGroup. By grouping the use cases and setting the viewport, CameraX ensures that the border rectangles of all use cases in the group point to the same camera sensor area. The following code snippet shows how to use these two classes: val viewPort = ViewPort.Builder(Rational(width, height), display.rotation).build() val useCaseGroup = UseCaseGroup.Builder() .addUseCase(preview) .addUseCase(imageAnalysis) .addUseCase(imageCapture) .setViewPort(viewPort) .build() cameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, useCaseGroup) ViewPort viewPort = new ViewPort.Builder(new Rational(width, height), getRotation(Display))). .build(); UseCaseGroup useCaseGroup = new UseCaseGroup.Builder() .addUseCase(preview) .addUseCase(imageAnalysis) .addUseCase(imageCapture) .setViewPort(viewPort) .build(); cameraProvider.bindToLifecycle(lifecycle owner, CameraSelector, useCaseGroup); The ViewPort defines a buffer rectangle that is visible to end users. CameraX then calculates the maximum possible crop rectangle based on the viewport properties and the attached use cases. To achieve a WYSIWYG effect, it is usually necessary to adjust the viewport according to the use case of the preview. An easy way to get viewports is to use the preview view. The following code snippets showto get the ViewPort object: val viewport = findViewById(R.id.preview_view).viewPort ViewPort viewPort = ((PreviewView)findViewById(R.id.preview_view)).getViewPort(); In the previous example, what the application receives from ImageAnalysis and ImageCapture will match what the end user sees in the PreviewView, provided the PreviewView zoom type is set to the default FILL_CENTER. After applying straight and rotated cropping to the output buffer, the image from all use cases will be the same, although possibly at different resolutions. For more information about applying transformation information, see Transformation Output. Camera Selection CameraX automatically selects the best camera based on your application requirements and use cases. If you want to use a device other than the selected device, there are several options: Note: Cameras must be recognized by the system and mapped to CameraManager.getCameraIdList() before they can be used. In addition, each OEM is responsible for selecting external camera support. So make sure PackageManager.FEATURE_CAMERA_EXTERNAL is enabled before trying to use external cameras. The following code example shows how to create a CameraSelector to select a device: fun selectExternalOrBestCamera(provider: ProcessCameraProvider):CameraSelector? { val cam2Infos = Vendor. availableCameraInfos.map { Camera2CameraInfo.from(it) }.sortedByDescending { // HARDWARE_LEVEL is of type Int, with order: // LEGACY < LIMITED < FULL < LEVEL_3 < EXTERNAL it.getCameraCharacteristic(CameraCharacteristics.INFO_SUPPORTED_HARDWARE)_LEVEL } return when { cam2Infos .isNotEmpty() -> { CameraSelector.Builder() .addCameraFilter { it.filter { camInfo -> // cam2Infos[0] EXTERNAL or better embedded camera val thisCamId = Camera2CameraInfo.from(camInfo) .cameraId thisCamId == cam2Infos[0].cameraId } }.build() } else -> null } } // Create a CameraSelector for the USB camera (or top-levelcamera) val selector = selectExternalOrBestCamera(processCameraProvider) processCameraProvider.bindToLifecycle(this, selector, preview, analyze) Camera resolution. You can allow CameraX to set the image resolution based on device performance, supported device hardware level, use case, and specified aspect ratio. Additionally, you can set a specific target resolution or aspect ratio for use cases that support this configuration. Automatic Resolution CameraX can automatically determine the best resolution settings based on the use cases specified in cameraProcessProvider.bindToLifecycle(). If possible, specify all use cases that need to run concurrently in the same session in a single bindToLifecycle() call. CameraX determines the resolution based on several use cases related to the hardware level supported by the device and for a specific device (where the device may or may not exceed the available stream configurations). The goal is to allow the application to run across devices while minimizing device-specific code paths. The standard aspect ratio for image capture and analysis is 4:3. Use cases have a configurable aspect ratio, so the application can specify the desired aspect ratio based on the UI design. CameraX outputs as accurately as the device supports, according to the required aspect ratio. If the exact match resolution is not supported, the one that matches the most conditions is chosen. So the app determines what the camera should look like in the app, and CameraX determines the best camera resolution settings to suit different devices. For example, an application might do the following: Specify a target resolution of 4:3 or 16:9 for a use case. Specify a custom resolution that CameraX will try to find the best match for. Specify the cropping aspect ratio for ImageCapture CameraX. select internal camera2resolutions automatically. The following table shows the resolutions: Use Case Internal Surface Resolution Output Resolution Preview Aspect Ratio: The resolution that best suits the purpose set. Internal surface resolution. Metadata is provided to allow the view to be cropped, scaled, and rotated to achieve the desired aspect ratio. Default Resolution: The highest preview resolution or the highest device-preferred resolution that matches the aspect ratio specified above. Maximum Resolution: The preview size, which refers to the size that best fits the device's screen resolution, or up to 1080p (1920x1080), whichever is smaller. Image Analysis Aspect Ratio: The resolution that best matches the set target. Internal surface resolution. Default Resolution: The default setting for the target resolution is 640x480. Adjusting both the target resolution and the corresponding aspect ratio gives the best supported resolution. MaxResolution: The maximum output resolution of the camera device in YUV_420_888 format, retrieved from StreamConfigurationMap.getOutputSizes(). The target resolution is set to 640 x 480 by default. So if you want a higher resolution than 640x480 you need to use setTargetResolution() and setTargetAspectRatio() to get the next supported resolution. Image Capture Aspect Ratio: The aspect ratio that best matches the setting. Internal surface resolution. Standard Resolution: The highest resolution available or supported by your device that conforms to the aspect ratio above. Max Resolution: The maximum output resolution of the camera device in JPEG format. Use StreamConfigurationMap.getOutputSizes() to load it. Setting the Resolution When building use cases, you can set specific resolutions using setTargetResolution(SizeResolution), as shown in the following code example: val imageAnalysis = ImageAnalysis.Builder() .setTargetResolution(Size(1280, 720)).build( ) ImageAnalysis imageAnalysis = new ImageAnalysis.Builder() .setTargetResolution(new Size(1280, 720)) .build(); Youset both target aspect ratio and target resolution for the same use case. Creating a configuration object will throw an IllegalArgumentException. Express the magnitude of the resolution in the coordinate system after rotating the supported dimensions with the target rotation. For example, a device with an initial vertical orientation in a natural target rotation that requires a vertical image may have a resolution of 480 x 640, and the same device rotated 90 degrees and oriented horizontally may have a resolution of 640 x 480. The target resolution tries to set the image resolution minimum threshold. The actual image resolution will be the closest available resolution that is not less than the target resolution specified by the camera implementation. However, if the resolution is not equal to or greater than the target resolution, the closest available resolution that is less than the target resolution is selected. Resolutions with the same aspect ratio of the specified size take precedence over resolutions with other aspect ratios. CameraX will use the best suitable resolution based on the requirements. If your main need is to adjust the aspect ratio, just type setTargetAspectRatio and CameraX will determine the specific resolution that is appropriate based on your device. If the application's primary need is to specify a resolution to increase image processing performance (for example, a small or medium image based on the device's processing capabilities), use setTargetResolution(Size resolution). Note. If you use the setTargetResolution() method, you can end up with a buffer whose aspect ratio doesn't match other use cases. If the aspect ratios must match, check the sizes of the returned buffers in both use cases and crop or resize one to match the other. If your application requires a precise resolution, see the table in createCaptureSession() to see what maximum resolutions are supported at each hardware level. To check the specific resolutions supported by the interfaceDevice, see StreamConfigurationMap.getOutputSizes(int). If your app is running Android 10 or later, you can use the isSessionConfigurationSupported() function to verify a specific session configuration. Camera Output Control In addition to being able to configure camera output as needed for each individual use case, CameraX also implements the following interface for controlling camera operations that are common to all related use cases: CameraControl allows you to configure common camera functions. You can use CameraInfo to find out the status of these general camera functions. Here are the camera functions supported in CameraControl: Zoom Focus and flashlight metering (click focus) Exposure compensation Get CameraControl and CameraInfo instances Get CameraControl and CameraInfo instances using the Camera object returned from ProcessCameraProvider.bindToLifecyle(). The following code shows an example: val camera = processCameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, preview) // Perform operations that affect all outputs. val cameraControl = camera.cameraControl // Check info and status. val cameraInfo = camera.cameraInfo Camera camera = processCameraProvider.bindToLifecycle(lifecycleOwner, cameraSelector, Preview) // Perform operations affecting all outputs. CameraControl cameraControl = camera.getCameraControl() // Check information and status. CameraInfo cameraInfo = camera.getCameraInfo() For example, zoom and other CameraControl operations can be passed by calling the bindToLifecycle() method. After exiting or destroying the activity used to bind the camera instance, the CameraControl can no longer perform the operation and returns a failed ListenableFuture. Note: After closing the camera, all changes to the zoom, flashlight, focus, metering status, and exposure compensation settings will return to their default values, i.e. H. when the lifecycle owner is stopped or destroyed. Zoom CameraControl offers two ways to change the zoom level:adjusts the scale according to the zoom factor. The ratio must be between CameraInfo.getZoomState().getValue().getMinZoomRatio() and CameraInfo.getZoomState().getValue().getMaxZoomRatio(). Otherwise, the function returns a ListenableFuture error. setLinearZoom() sets the current zoom to a linear zoom value between 0 and 1.0. The advantage of linear zoom is that it allows the field of view (FOV) to scale as the zoom changes. This makes it ideal for use with a slider. CameraInfo.getZoomState() returns the LiveData of the current zoom state. The value is changed when the camera is initialized or when the zoom level is set using setZoomRatio() or setLinearZoom(). Calling any method sets the values that ZoomState.getZoomRatio() and ZoomState.getLinearZoom() support. This is useful if you want to display zoom factor text next to the slider. Just look at ZoomState LiveData to update both without conversion. The ListenableFuture returned by both APIs allows applications to be notified when a request is retried at the specified scale value. Also, if you set a new zoom value while the previous activity is still running, the previous zoom activities in the ListenableFuture will fail immediately. Torch CameraControl.enableTorch(bool) enables or disables the flashlight (also known as flash). The current state of the torch can be found using CameraInfo.getTorchState(). You can check the value returned by CameraInfo.hasFlashUnit() to see if flash is available. Otherwise, calling CameraControl.enableTorch(boolean) immediately terminates the returned ListenableFuture with an erroneous result and sets the Torch state to TorchState.OFF. When the flashlight is on, it stays on when taking photos and videos, regardless of the flash mode setting. FlashMode in ImageCapture only works when flash is disabled. Focus and Metering CameraControl.startFocusAndMetering() activates autofocus and metering by settingMetering areas based on the given FocusMeteringAction. This is commonly used to implement the "tap to focus" feature in many camera applications. MeteringPoint First create a MeteringPoint with MeteringPointFactory.createPoint(float x, float y, float size). The measuring point is a single point on the surface of the chamber. It is stored in a normalized form so it can be easily converted to sensor coordinates to define AF/AE/AWB areas. The MeteringPoint size ranges from 0 to 1, the default size is 0.15f. When you call MeteringPointFactory.createPoint(float x, float y, float size), CameraX creates a rectangular area centered at (x, y) for the specified size. The following code shows how to create a MeteringPoint: // Use PreviewView.getMeteringPointFactory if PreviewView is being used for previewing. previewView.setOnTouchListener((view, motionEvent) -> { valmeteringPoint = previewView.meteringPointFactory .createPoint(motionEvent.x, motionEvent.y) … } // Uses DisplayOrientedMeteringPointFactory if SurfaceView / TextureView is // used for previewing. Note that if the preview is scaled or clipped in the view, // the application is responsible for properly transforming the coordinates so // that the width and height of this factory represent the full preview box. // The A(x, y) passed to create the MeteringPoint may need to be modified with //offsets ImageAnalysis ImageProxy.val MeteringPointFactory = SurfaceOrientedMeteringPointFactory(imageWidth, imageHeight, imageAnalysis) startFocusAndMetering and FocusMeteringAction Multiple applications need to be created to call the startFocusAndMetering and Metering(MeteringAction) applications Combination of cs measurement modes from FLAG_AF, FLAG_AE,The following code illustrates this usage: val MeteringPoint1 = MeteringPointFactory.createPoint(x1, x1) val MeteringPoint2 = MeteringPointFactory.createPoint(x2, y2) val action = FocusMeteringAction.Builder(meteringPoint1) // Default AWAF //E| connect MeteringPoint2 AF/AE. .addPoint(meteringPoint2, FLAG_AF | FLAG_AE) // Action will be canceled after 3 seconds (if not set, default is 5 seconds). .setAutoCancelDuration(3, TimeUnit.SECONDS) .build() val result = cameraControl.startFocusAndMetering(action) // Add a listener to ListenableFuture if you want to know the focus metering result. result.addListener({ //result.get().isFocusSuccessful returns whether autofocus worked or not. }, ContextCompat.getMainExecutor(this) As shown in the code above, startFocusAndMetering() performs a FocusMeteringAction consisting of a single metering For AF point metering. /AE/AWB area and another metering point for AF and AE only. Internally CameraX converts it to Camera2 MeteringRectanges and sets appropriate CONTROL_AF_REGIONS / CONTROL_AE_REGIONS / CONTROL_AWB_REGIONS to request capture. Since not every device supports AF/AE/AWB and idle areas, the camera with maximum impact CameraX will use the maximum number of metering points supported in the order in which they are added All metering points added after the maximum number are ignored For example, if a FocusMeteringAction has 3 metering points on a platform that only supports 2, only P the first 2 measurement points.the measurement point is ignored by ez CameraX.Exposure compensation Exposure compensation is useful when applications require fine-tuning of exposure (EV) values outside of automatic exposure (AE) output. The exposure compensation values are combined as follows to determine the appropriate exposure for the current image conditions: Exposure = Exposure Compensation Index * Exposure Compensation Stepfunction to set the exposure compensation as an index value. Positive values of the index make the image brighter, and negative values make the image darker. Applications can query the supported range using CameraInfo.ExposureState.exposureCompensationRange(), which is described in the next section. If the value is supported, the returned ListenableFuture completes when the value is successfully included in the capture request; if the specified index is outside the supported range, setExposureCompensationIndex() causes the returned ListenableFuture to exit immediately with an error result. CameraX only stores the last outstanding setExposureCompensationIndex() request, and calling the function repeatedly before the previous request is completed will cancel it. This snippet sets the exposure compensation index and registers a callback when an exposure change is requested: camera.cameraControl.setExposureCompensationIndex(exposureCompensationIndex) .addListener({ // Get the current exposure compensation index, it may // differ from the desired value if this request // the newer setting request is canceled val currentExposureIndex = camera.cameraInfo.exposureState.exposureCompensationIndex ¦ }, mainExecutor) Camera.CameraInfo.getExposureState() retrieves the current ExposureState, including: Option to control exposure compensation. Current exposure compensation index. Range of exposure compensation index. The exposure compensation step used to calculate the exposure compensation value. For example, the following code initializes the exposure settings on the SeekBar with the current ExposureState values: bottom progress =exposureState.exposureCompensationIndex } More Exploration Resourcesfor CameraX, see the additional resources below. Getting Started with Codelab with CameraX Sample Code CameraX Sample Apps Android Developer Community CameraX Discussion Group