Камера

Одно из преимуществ смартфонов – наличие камер, которые можно использовать для получения изображений внешнего мира.

В настоящее время основной рекомендуемый способ работы с камерой – использование библиотеки CameraX. Она основана на другой библиотеке Camera2, которая предоставляет более низкоуровневый доступ к камере и позволяет контроллировать каждую деталь, однако такие возможности требутся нечасто, поэтому в большинстве случаев возможностей библиотеки CameraX вполне достаточно.

Библиотека CameraX не включена в SDK по умолчанию, для её использования необходимо добавить следующие ссылки в файл скриптов Gradle для модуля:

```
implementation("androidx.camera:camera-core:1.5.0-alpha01")
implementation("androidx.camera:camera-camera2:1.5.0-alpha01")
implementation("androidx.camera:camera-lifecycle:1.5.0-alpha01")
implementation("androidx.camera:camera-view:1.5.0-alpha01")
implementation("androidx.camera:camera-extensions:1.5.0-alpha01")
```

Поскольку работа с камерой затрагивает конфиденциальность пользователя, необходимо запрашивать разрешение у пользователя. Первое, что требуется сделать – это добавить в манифест строки с разрешением `CAMERA` и с требованием её наличия:

```
<uses-feature
    android:name="android.hardware.camera"
    android:required="true" />
<uses-permission
    android:name="android.permission.CAMERA"/>
```

Если функционал приложения полностью зависит от наличия камеры, свойство `required` должно иметь значение `true`, в таком случае приложение не сможет быть установлено на устройство, в котором камера отсутствует. Если использование камеры является лишь одной из функций приложения, можно поставить значение `false`.

Помимо изменений в манифесте, нужно произвести запрос разрешения в коде приложения. Дальнейшие действия предполагают что разрешение на использование камеры приложением получено.

Классы для работы с камерой

Основным классом, который предоставляет доступ к камере, является `CameraProvider`. Поскольку инициализация оборудования может потребовать времени, создавать объект этого класса следует не напрямую, а в «отложенном режиме». Для этого в Android используется класс `ListenableFuture`, который производит инициализацию объекта, а по её окончании вызывает срабатывание слушателя. Размещают такой объект обычно в классе активности:

```
private lateinit var cameraProviderFuture: ListenableFuture<ProcessCameraProvider>
```

Когда требуется начать работу с камерой (например, после получения разрешения от пользователя), нужно запросить экземпляр этого объекта, а всю дальнейшую инициализацию вынести в код слушателя:

```
cameraProviderFuture = ProcessCameraProvider.getInstance(this)
cameraProviderFuture.addListener({
    val cameraProvider = cameraProviderFuture.get()
    // Дальнейшая инициализация камеры
}, ContextCompat.getMainExecutor(this))
```

При работе с камерой пользователю обычно демонстрируется то, что камера в данный момент «видит». Поскольку камера предоставляет поток таких изображений с частотой от 30 кадров в секунду и выше, требуется их быстрая отрисовка на экране. Для этого хорошо подходит класс `Surface` и его наследники, работа с ними кратко описана в теме «Рисование».

Android предоставляет элемент управления `PreviewView`, который можно разместить на экране и использовать для отрисовки потока изображений с камеры. Обычно этот элемент занимает всю поверхность экрана, а другие элементы располагаются над ним, поэтому имеет смысл использовать контейнер `FrameLayout`:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.camera.view.PreviewView
        android:id="@+id/preview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="fill"/>

    <ImageButton
        android:id="@+id/capture"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_marginBottom="24dp"
        android:layout_marginEnd="24dp"/>
    ...
</FrameLayout>
```

Подготовка объектов

Для работы объект `cameraProvider` требует подключения нескольких дополнительных объектов:

- `Preview` для отображения потока изображений с камеры на экране
- `CameraSelector` для выбора конкретной камеры из нескольких
- `ImageCapture` для захвата кадра
- `VideoCapture` для захвата видео
- `ImageAnalysis` для анализа потока изображений с камеры «на лету»

Каждый такой объект создаётся отдельно, а потом подключается к объекту `cameraProvider`.

Класс Preview

Для отображения потока изображений с камеры нужно создать связь объекта `cameraProvider` и элемента `PreviewView`, для этого используется класс-посредник `Preview`:

```
val preview = Preview.Builder()
    .build()
preview.surfaceProvider = pvPreview.getSurfaceProvider()
```

Здесь и далее `pvPreview` – это ссылка на элемент управления `PreviewView`, полученная в коде программы.

Класс CameraSelector

Также следует выбрать с какой именно камеры будет осуществляться получение изображений, для этого используется класс `CameraSelector`:

```
val cameraSelector = CameraSelector.Builder()
    .requireLensFacing(CameraSelector.DEFAULT_BACK_CAMERA)
    .build()
```

В этом примере выбраны задние камеры (`DEFAULT_BACK_CAMERA`), можно выбрать фронтальную камеру (`DEFAULT_FRONT_CAMERA`).

Класс ImageCapture

Для захвата кадра используется класс `ImageCapture`:

```
val imageCapture = ImageCapture.Builder()
    .setTargetRotation(pvPreview.display.rotation)
    .build()
```

В этом коде задаётся поворот снимка – если телефон находится не в портретной ориентации, то и снимок тоже будет в аналогичной ориентации.

Объект `imageCapture` позже будет использоваться для захвата кадра, поэтому ссылку на него следует сохранить.

Подключение объектов

Теперь, когда в программе есть объекты `preview` для организации просмотра потока изображений, `cameraSelector` для выбора конкретной камеры, и `imageCapture` для захвата кадра, нужно подключить их все к объекту `cameraProvider`, который будет использовать эти объекты в работе:

```
cameraProvider.bindToLifecycle(this,
    cameraSelector,
    imageCapture,
    preview)
```

Объект `cameraProvider` привязывается к жизненному циклу активности или фрагмента (на который указывает ссылка `this`), что позволяет автоматически включать или отключать камеру когда активность или фрагмент становятся активными ли неактивными. В прошлом разработчику требовалось самому отслеживать эти события.

Захват кадра

Когда пользователь решает, что изображение на экране содержит подходящий кадр, он может дать программе указание произвести его захват («сделать снимок»). Обычно для этого на экране нажимается кнопка с изображением камеры.

Непосредственно процесс захвата кадра осуществляется с помощью вызова метода `takePicture` у объекта `imageCapture`. Поскольку процесс захвата может быть длительным, он будет выполнен в фоновом режиме, поэтому в качестве одного из параметров передаётся объект `Executor`, который и запускает код в нужном потоке. Другой параметр – это слушатель, который будет вызван после захвата кадра (или при ошибке захвата):

```
imageCapture.takePicture(
    outputFileOptions,
    ContextCompat.getMainExecutor(this),
    object : ImageCapture.OnImageSavedCallback {
        override fun onError(error: ImageCaptureException)
        {
            // Ошибка при захвате кадра
        }
        override fun onImageSaved(outputFileResults: ImageCapture.OutputFileResults) {
            // Успешный захват кадра
        }
    })
```

Объект `outputFileOptions` – это ссылка на файл, в который будет записан файл. Если требуется сохранить фотографию в частную папку приложения, то можно использовать функцию `openFileOutput`, а если нужно сохранить фотографию в публичный каталог с фотографиями камеры, то код будет примерно таким:

```
val f = File(
    Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DCIM),
    "photo.jpg")
val outputFileOptions = ImageCapture.OutputFileOptions.Builder(f).build()
```

Анализ кадра

Иногда во время предварительного просмотра может быть полезно выполнить какой-либо анализ потока изображений, поступающих с камеры: например, для подбора подходящей выдержки снимка или определения факта наличия в кадре каких-либо объектов. Анализатор создаётся так же, как и другие объекты для работы с камерой:

```
val imageAnalysis = ImageAnalysis.Builder()
    .setOutputImageFormat(ImageAnalysis.OUTPUT_IMAGE_FORMAT_RGBA_8888)
    .build()

imageAnalysis.setAnalyzer(ContextCompat.getMainExecutor(this)) { imageProxy ->
    val buffer = imageProxy.planes[0].buffer
    buffer.rewind()
    // Анализ изображения
    // ...
    imageProxy.close()
}
```

При создании объекта устанавливается режим RGBA_8888, чтобы каждый пиксель был представлен в формате RGBA, а не в обычном для камеры YUV. Каждое изображение из потока передаётся в анализатор параметром `imageProxy`. Приложение должно быстро его проанализировать и при необходимости сделать какие-то изменения в дальнейшей работе. Анализ не должен быть долгим и ресурсоёмким, поскольку в этом случае кадры не будут успевать обрабатываться.

Буфер с данными `buffer`, который получается из `imageProxy` – это объект класса `ByteBuffer`, он содержит изображение в виде одномерного массива, каждый пиксель которого представлен в виде четырёх байт, соответствующих компонентам R, G, B и A:

Смещение	Строка	Столбец	Описание
0	0	0	Компонент R
1	0	0	Компонент G
2	0	0	Компонент B
3	0	0	Компонент A
4	0	1	Компонент R
5	0	1	Компонент G
6	0	1	Компонент B
7	0	1	Компонент A
8	0	2	Компонент R
…			

Когда заканчиваются байты, относящиеся к 0-й строке, сразу же начинаются байты, относящиеся к 1-й строке, и т. д. Таким образом, смещение нужного пикселя (точнее, его компонента R) можно рассчитать по следующей формуле:

$$\text{Смещение} = (\text{Строка} \cdot \text{КоличествоСтолбцов} + \text{Столбец}) \cdot 4$$

Размер изображения можно узнать с помощью свойств `imageProxy.width` и `imageProxy.height`. Например, если размер изображения 640×480, то количество пикселей будет 307’200, а размер буфера в четыре раза больше – 1’228’800 байт. Чтобы получить компоненты пикселя с координатами [x,y] можно использовать примерно такой код:

```
val offset = (y * imageProxy.width + x) * 4
val r = buffer[offset + 0].toInt() and 0xff
val g = buffer[offset + 1].toInt() and 0xff
val b = buffer[offset + 2].toInt() and 0xff
```

После создания анализатора его нужно подключить к объекту `cameraProvider` вместе с остальными объектами:

```
cameraProvider.bindToLifecycle(this,
    cameraSelector,
    imageCapture,
    imageAnalysis,
    preview)
```

Задание

Разработайте приложение для работы с камерой. Приложение должно быть основано на библиотеке CameraX. В нём должен быть реализован следующий функционал:

- Определение HTML-цвета точки в центре снимка. В центре экрана нужно разместить визуальный объект – например, круг или пересечение линий – который показывает пользователю примерное место размещения этого пикселя. Когда пользователь перемещает камеру, приложение постоянно анализирует цвет центрального пикселя и выводит его HTML-код на экран.
- При нажатии кнопки съемки осуществляется захват и сохранение текущего кадра в галерею телефона.

Приложение может выглядеть примерно таким образом:

Чтобы сделать круглый «прицел», можно добавить XML-файл crosshair.xml в раздел res → drawable со следующим содержимым:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">

    <solid
        android:color="#00000000"/>
    <stroke
        android:color="@color/white"
        android:width="1dp"/>
    <size
        android:width="7dp"
        android:height="7dp"/>

</shape>
```

Здесь у элемента `shape` задаётся форма `oval`, а вложенные элементы задают параметры: `solid` – цвет (в данном случае прозрачный), `stroke` – обводку, `size` – размер. Затем в разметку активности можно добавить элемент `ImageView` с такими параметрами:

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/crosshair"
    android:layout_gravity="center"/>
```

Это разместит элемент-«прицел» по центру экрана.