Camera API

In this document

Key classes

- `Camera`
- `SurfaceView`
- `MediaRecorder`
- `Intent`

See also

- MediaPlayer
- Data Storage

The Android framework includes support for various cameras and camera features available on devices, allowing you to capture pictures and videos in your applications. This document discusses a quick, simple approach to image and video capture and outlines an advanced approach for creating custom camera experiences for your users.

> **Note:** This page describes the `Camera` class, which has been deprecated. We recommend using the newer class `camera2`, which works on Android 5.0 (API level 21) or greater. Read more about camera2 on our blog.

Considerations

Before enabling your application to use cameras on Android devices, you should consider a few questions about how your app intends to use this hardware feature.

- **Camera Requirement** - Is the use of a camera so important to your application that you do not want your application installed on a device

that does not have a camera? If so, you should declare the camera requirement in your manifest.

- **Quick Picture or Customized Camera** - How will your application use the camera? Are you just interested in snapping a quick picture or video clip, or will your application provide a new way to use cameras? For a getting a quick snap or clip, consider Using Existing Camera Apps. For developing a customized camera feature, check out the Building a Camera App section.
- **Storage** - Are the images or videos your application generates intended to be only visible to your application or shared so that other applications such as Gallery or other media and social apps can use them? Do you want the pictures and videos to be available even if your application is uninstalled? Check out the Saving Media Files section to see how to implement these options.

The basics

The Android framework supports capturing images and video through the `android.hardware.camera2` API or camera `Intent`. Here are the relevant classes:

`android.hardware.camera2`

> This package is the primary API for controlling device cameras. It can be used to take pictures or videos when you are building a camera application.

`Camera`

> This class is the older deprecated API for controlling device cameras.

`SurfaceView`

> This class is used to present a live camera preview to the user.

`MediaRecorder`

> This class is used to record video from the camera.

`Intent`

> An intent action type of `MediaStore.ACTION_IMAGE_CAPTURE` or `MediaStore.ACTION_VIDEO_CAPTURE` can be used to capture images or videos without directly using the `Camera` object.

Manifest declarations

Before starting development on your application with the Camera API, you should make sure your manifest has the appropriate declarations to allow use of camera hardware and other related features.

- **Camera Permission** - Your application must request permission to use a device camera.

```
<uses-permission android:name="android.permission.CAMERA" />
```

> **Note:** If you are using the camera by invoking an existing camera app, your application does not need to request this permission.

- **Camera Features** - Your application must also declare use of camera features, for example:

```
<uses-feature android:name="android.hardware.camera" />
```

For a list of camera features, see the manifest Features Reference.

Adding camera features to your manifest causes Google Play to prevent your application from being installed to devices that do not include a camera or do not support the camera features you specify. For more information about using feature-based filtering with Google Play, see Google Play and Feature-Based Filtering.

If your application *can use* a camera or camera feature for proper operation, but does not *require* it, you should specify this in the manifest by including the `android:required` attribute, and setting it to `false`:

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

- **Storage Permission** - If your application saves images or videos to the device's external storage (SD Card), you must also specify this in the manifest.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

- **Audio Recording Permission** - For recording audio with video capture, your application must request the audio capture permission.

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

- **Location Permission** - If your application tags images with GPS location information, you must request the `ACCESS_FINE_LOCATION` permission. Note that, if your app targets Android 5.0 (API level 21) or higher, you also need to declare that your app uses the device's GPS:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
...
<!-- Needed only if your app targets Android 5.0 (API level 21) or higher. -->
<uses-feature android:name="android.hardware.location.gps" />
```

For more information about getting user location, see Location Strategies.

Using existing camera apps

A quick way to enable taking pictures or videos in your application without a lot of extra code is to use an `Intent` to invoke an existing Android camera application. The details are described in the training lessons Taking Photos Simply and Recording Videos Simply.

Building a camera app

Some developers may require a camera user interface that is customized to the look of their application or provides special features. Writing your own picture-taking code can provide a more compelling experience for your users.

Note: The following guide is for the older, deprecated `Camera` API. For new or advanced camera applications, the newer `android.hardware.camera2` API is recommended.

The general steps for creating a custom camera interface for your application are as follows:

- **Detect and Access Camera** - Create code to check for the existence of cameras and request access.
- **Create a Preview Class** - Create a camera preview class that extends `SurfaceView` and implements the `SurfaceHolder` interface. This class previews the live images from the camera.
- **Build a Preview Layout** - Once you have the camera preview class, create a view layout that incorporates the preview and the user interface controls you want.
- **Setup Listeners for Capture** - Connect listeners for your interface controls to start image or video capture in response to user actions, such as pressing a button.
- **Capture and Save Files** - Setup the code for capturing pictures or videos and saving the output.
- **Release the Camera** - After using the camera, your application must properly release it for use by other applications.

Camera hardware is a shared resource that must be carefully managed so your application does not collide with other applications that may also want to use it. The following sections discusses how to detect camera hardware, how to request access to a camera, how to capture pictures or video and how to release the camera when your application is done using it.

> **Caution:** Remember to release the `Camera` object by calling the `Camera.release()` when your application is done using it! If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down.

Detecting camera hardware

If your application does not specifically require a camera using a manifest declaration, you should check to see if a camera is available at runtime. To perform this check, use the `PackageManager.hasSystemFeature()` method, as shown in the example code below:

```
/** Check if this device has a camera */
private boolean checkCameraHardware(Context context) {
    if (context.getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA)){
        // this device has a camera
        return true;
    } else {
        // no camera on this device
        return false;
    }
}
```

Android devices can have multiple cameras, for example a back-facing camera for photography and a front-facing camera for video calls. Android 2.3 (API Level 9) and later allows you to check the number of cameras available on a device using the `Camera.getNumberOfCameras()` method.

Accessing cameras

If you have determined that the device on which your application is running has a camera, you must request to access it by getting an instance of `Camera` (unless you are using an intent to access the camera).

To access the primary camera, use the `Camera.open()` method and be sure to catch any exceptions, as shown in the code below:

```
/** A safe way to get an instance of the Camera object. */
public static Camera getCameraInstance(){
    Camera c = null;
    try {
        c = Camera.open(); // attempt to get a Camera instance
    }
    catch (Exception e){
        // Camera is not available (in use or does not exist)
    }
    return c; // returns null if camera is unavailable
}
```

> **Caution:** Always check for exceptions when using `Camera.open()`. Failing to check for exceptions if the camera is in use or does not exist will cause your application to be shut down by the system.

On devices running Android 2.3 (API Level 9) or higher, you can access specific cameras using `Camera.open(int)`. The example code above will access the first, back-facing camera on a device with more than one camera.

Checking camera features

Once you obtain access to a camera, you can get further information about its capabilities using the `Camera.getParameters()` method and checking the returned `Camera.Parameters` object for supported capabilities. When using API Level 9 or higher, use the `Camera.getCameraInfo()` to determine if a camera is on the front or back of the device, and the orientation of the image.

Creating a preview class

For users to effectively take pictures or video, they must be able to see what the device camera sees. A camera preview class is a `SurfaceView` that can display the live image data coming from a camera, so users can frame and capture a picture or video.

The following example code demonstrates how to create a basic camera preview class that can be included in a `View` layout. This class implements `SurfaceHolder.Callback` in order to capture the callback events for creating and destroying the view, which are needed for assigning the camera preview input.

```
/** A basic Camera preview class */
public class CameraPreview extends SurfaceView implements SurfaceHolder.Callback {
    private SurfaceHolder mHolder;
    private Camera mCamera;

    public CameraPreview(Context context, Camera camera) {
        super(context);
        mCamera = camera;

        // Install a SurfaceHolder.Callback so we get notified when the
        // underlying surface is created and destroyed.
        mHolder = getHolder();
        mHolder.addCallback(this);
        // deprecated setting, but required on Android versions prior to 3.0
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }

    public void surfaceCreated(SurfaceHolder holder) {
        // The Surface has been created, now tell the camera where to draw the preview.
        try {
            mCamera.setPreviewDisplay(holder);
            mCamera.startPreview();
        } catch (IOException e) {
            Log.d(TAG, "Error setting camera preview: " + e.getMessage());
        }
    }

    public void surfaceDestroyed(SurfaceHolder holder) {
        // empty. Take care of releasing the Camera preview in your activity.
    }

    public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
        // If your preview can change or rotate, take care of those events here.
        // Make sure to stop the preview before resizing or reformatting it.

        if (mHolder.getSurface() == null){
          // preview surface does not exist
          return;
        }

        // stop preview before making changes
        try {
            mCamera.stopPreview();
        } catch (Exception e){
          // ignore: tried to stop a non-existent preview
        }

        // set preview size and make any resize, rotate or
        // reformatting changes here

        // start preview with new settings
        try {
            mCamera.setPreviewDisplay(mHolder);
            mCamera.startPreview();

        } catch (Exception e){
            Log.d(TAG, "Error starting camera preview: " + e.getMessage());
        }
    }
}
```

If you want to set a specific size for your camera preview, set this in the `surfaceChanged()` method as noted in the comments above. When setting preview size, you *must use* values from `getSupportedPreviewSizes()`. *Do not* set arbitrary values in the `setPreviewSize()` method.

> **Note:** With the introduction of the Multi-Window feature in Android 7.0 (API level 24) and higher, you can no longer assume the aspect ratio of the preview is the same as your activity even after calling `setDisplayOrientation()`. Depending on the window size and aspect ratio, you may may have to fit a wide camera preview into a portrait-orientated layout, or vice versa, using a letterbox layout.

Placing preview in a layout

A camera preview class, such as the example shown in the previous section, must be placed in the layout of an activity along with other user interface controls for taking a picture or video. This section shows you how to build a basic layout and activity for the preview.

The following layout code provides a very basic view that can be used to display a camera preview. In this example, the `FrameLayout` element is meant to be the container for the camera preview class. This layout type is used so that additional picture information or controls can be overlaid on the live camera preview images.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <FrameLayout
    android:id="@+id/camera_preview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    />

  <Button
    android:id="@+id/button_capture"
    android:text="Capture"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    />
</LinearLayout>
```

On most devices, the default orientation of the camera preview is landscape. This example layout specifies a horizontal (landscape) layout and the code below fixes the orientation of the application to landscape. For simplicity in rendering a camera preview, you should change your application's preview activity orientation to landscape by adding the following to your manifest.

```
<activity android:name=".CameraActivity"
          android:label="@string/app_name"

          android:screenOrientation="landscape">
          <!-- configure this activity to use landscape orientation -->

          <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

> **Note:** A camera preview does not have to be in landscape mode. Starting in Android 2.2 (API Level 8), you can use the `setDisplayOrientation()` method to set the rotation of the preview image. In order to change preview orientation as the user re-orients the phone, within the `surfaceChanged()` method of your preview class, first stop the preview with `Camera.stopPreview()` change the orientation and then start the preview again with `Camera.startPreview()`.

In the activity for your camera view, add your preview class to the `FrameLayout` element shown in the example above. Your camera activity must also ensure that it releases the camera when it is paused or shut down. The following example shows how to modify a camera activity to attach the preview class shown in Creating a preview class.

```
public class CameraActivity extends Activity {

    private Camera mCamera;
    private CameraPreview mPreview;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Create an instance of Camera
        mCamera = getCameraInstance();

        // Create our Preview view and set it as the content of our activity.
        mPreview = new CameraPreview(this, mCamera);
        FrameLayout preview = (FrameLayout) findViewById(R.id.camera_preview);
        preview.addView(mPreview);
    }
}
```

> **Note:** The `getCameraInstance()` method in the example above refers to the example method shown in Accessing cameras.

Capturing pictures

Once you have built a preview class and a view layout in which to display it, you are ready to start capturing images with your application. In your application code, you must set up listeners for your user interface controls to respond to a user action by taking a picture.

In order to retrieve a picture, use the `Camera.takePicture()` method. This method takes three parameters which receive data from the camera. In order to receive data in a JPEG format, you must implement an `Camera.PictureCallback` interface to receive the image data and write it to a file. The following code shows a basic implementation of the `Camera.PictureCallback` interface to save an image received from the camera.

```
private PictureCallback mPicture = new PictureCallback() {

    @Override
    public void onPictureTaken(byte[] data, Camera camera) {

        File pictureFile = getOutputMediaFile(MEDIA_TYPE_IMAGE);
        if (pictureFile == null){
            Log.d(TAG, "Error creating media file, check storage permissions: " +
                e.getMessage());
            return;
        }

        try {
            FileOutputStream fos = new FileOutputStream(pictureFile);
            fos.write(data);
            fos.close();
        } catch (FileNotFoundException e) {
            Log.d(TAG, "File not found: " + e.getMessage());
        } catch (IOException e) {
            Log.d(TAG, "Error accessing file: " + e.getMessage());
        }
    }
};
```

Trigger capturing an image by calling the `Camera.takePicture()` method. The following example code shows how to call this method from a button `View.OnClickListener`.

```
// Add a listener to the Capture button
Button captureButton = (Button) findViewById(id.button_capture);
captureButton.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // get an image from the camera
            mCamera.takePicture(null, null, mPicture);
        }
    }
);
```

> **Note:** The `mPicture` member in the following example refers to the example code above.

> **Caution:** Remember to release the `Camera` object by calling the `Camera.release()` when your application is done using it! For information about how to release the camera, see Releasing the camera.

Capturing videos

Video capture using the Android framework requires careful management of the `Camera` object and coordination with the `MediaRecorder` class. When recording video with `Camera`, you must manage the `Camera.lock()` and `Camera.unlock()` calls to allow `MediaRecorder` access to the camera hardware, in addition to the `Camera.open()` and `Camera.release()` calls.

> **Note:** Starting with Android 4.0 (API level 14), the `Camera.lock()` and `Camera.unlock()` calls are managed for you automatically.

Unlike taking pictures with a device camera, capturing video requires a very particular call order. You must follow a specific order of execution to successfully prepare for and capture video with your application, as detailed below.

1. **Open Camera** - Use the `Camera.open()` to get an instance of the camera object.
2. **Connect Preview** - Prepare a live camera image preview by connecting a `SurfaceView` to the camera using `Camera.setPreviewDisplay()`.
3. **Start Preview** - Call `Camera.startPreview()` to begin displaying the live camera images.
4. **Start Recording Video** - The following steps must be completed *in order* to successfully record video:
    a. **Unlock the Camera** - Unlock the camera for use by `MediaRecorder` by calling `Camera.unlock()`.
    b. **Configure MediaRecorder** - Call in the following `MediaRecorder` methods *in this order*. For more information, see the `MediaRecorder` reference documentation.
        1. `setCamera()` - Set the camera to be used for video capture, use your application's current instance of `Camera`.
        2. `setAudioSource()` - Set the audio source, use `MediaRecorder.AudioSource.CAMCORDER`.
        3. `setVideoSource()` - Set the video source, use `MediaRecorder.VideoSource.CAMERA`.
        4. Set the video output format and encoding. For Android 2.2 (API Level 8) and higher, use the `MediaRecorder.setProfile` method, and get a profile instance using `CamcorderProfile.get()`. For versions of Android prior to 2.2, you must set the video output format and encoding parameters:
            i. `setOutputFormat()` - Set the output format, specify the default setting or `MediaRecorder.OutputFormat.MPEG_4`.
            ii. `setAudioEncoder()` - Set the sound encoding type, specify the default setting or `MediaRecorder.AudioEncoder.AMR_NB`.
            iii. `setVideoEncoder()` - Set the video encoding type, specify the default setting or `MediaRecorder.VideoEncoder.MPEG_4_SP`.
        5. `setOutputFile()` - Set the output file, use `getOutputMediaFile(MEDIA_TYPE_VIDEO).toString()` from the example method in the Saving Media Files section.
        6. `setPreviewDisplay()` - Specify the `SurfaceView` preview layout element for your application. Use the same object you specified for **Connect Preview**.

    > **Caution:** You must call these `MediaRecorder` configuration methods *in this order*, otherwise your application will encounter errors

> and the recording will fail.

c. **Prepare MediaRecorder** - Prepare the `MediaRecorder` with provided configuration settings by calling `MediaRecorder.prepare()`.

d. **Start MediaRecorder** - Start recording video by calling `MediaRecorder.start()`.

5. **Stop Recording Video** - Call the following methods *in order*, to successfully complete a video recording:

   a. **Stop MediaRecorder** - Stop recording video by calling `MediaRecorder.stop()`.

   b. **Reset MediaRecorder** - Optionally, remove the configuration settings from the recorder by calling `MediaRecorder.reset()`.

   c. **Release MediaRecorder** - Release the `MediaRecorder` by calling `MediaRecorder.release()`.

   d. **Lock the Camera** - Lock the camera so that future `MediaRecorder` sessions can use it by calling `Camera.lock()`. Starting with Android 4.0 (API level 14), this call is not required unless the `MediaRecorder.prepare()` call fails.

6. **Stop the Preview** - When your activity has finished using the camera, stop the preview using `Camera.stopPreview()`.

7. **Release Camera** - Release the camera so that other applications can use it by calling `Camera.release()`.

> **Note:** It is possible to use `MediaRecorder` without creating a camera preview first and skip the first few steps of this process. However, since users typically prefer to see a preview before starting a recording, that process is not discussed here.

> **Tip:** If your application is typically used for recording video, set `setRecordingHint(boolean)` to `true` prior to starting your preview. This setting can help reduce the time it takes to start recording.

Configuring MediaRecorder

When using the `MediaRecorder` class to record video, you must perform configuration steps in a *specific order* and then call the `MediaRecorder.prepare()` method to check and implement the configuration. The following example code demonstrates how to properly configure and prepare the `MediaRecorder` class for video recording.

```
private boolean prepareVideoRecorder(){

    mCamera = getCameraInstance();
    mMediaRecorder = new MediaRecorder();

    // Step 1: Unlock and set camera to MediaRecorder
    mCamera.unlock();
    mMediaRecorder.setCamera(mCamera);

    // Step 2: Set sources
    mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
    mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

    // Step 3: Set a CamcorderProfile (requires API Level 8 or higher)
    mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_HIGH));

    // Step 4: Set output file
    mMediaRecorder.setOutputFile(getOutputMediaFile(MEDIA_TYPE_VIDEO).toString());

    // Step 5: Set the preview output
    mMediaRecorder.setPreviewDisplay(mPreview.getHolder().getSurface());

    // Step 6: Prepare configured MediaRecorder
    try {
        mMediaRecorder.prepare();
    } catch (IllegalStateException e) {
        Log.d(TAG, "IllegalStateException preparing MediaRecorder: " + e.getMessage());
        releaseMediaRecorder();
        return false;
    } catch (IOException e) {
        Log.d(TAG, "IOException preparing MediaRecorder: " + e.getMessage());
        releaseMediaRecorder();
        return false;
    }
    return true;
}
```

Prior to Android 2.2 (API Level 8), you must set the output format and encoding formats parameters directly, instead of using `CamcorderProfile`. This approach is demonstrated in the following code:

```
    // Step 3: Set output format and encoding (for versions prior to API Level 8)
    mMediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
    mMediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
    mMediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.DEFAULT);
```

The following video recording parameters for `MediaRecorder` are given default settings, however, you may want to adjust these settings for your application:

- `setVideoEncodingBitRate()`
- `setVideoSize()`
- `setVideoFrameRate()`
- `setAudioEncodingBitRate()`
- `setAudioChannels()`
- `setAudioSamplingRate()`

Starting and stopping MediaRecorder

When starting and stopping video recording using the `MediaRecorder` class, you must follow a specific order, as listed below.

1. Unlock the camera with `Camera.unlock()`
2. Configure `MediaRecorder` as shown in the code example above
3. Start recording using `MediaRecorder.start()`

4. Record the video
5. Stop recording using `MediaRecorder.stop()`
6. Release the media recorder with `MediaRecorder.release()`
7. Lock the camera using `Camera.lock()`

The following example code demonstrates how to wire up a button to properly start and stop video recording using the camera and the `MediaRecorder` class.

> **Note:** When completing a video recording, do not release the camera or else your preview will be stopped.

```
private boolean isRecording = false;

// Add a listener to the Capture button
Button captureButton = (Button) findViewById(id.button_capture);
captureButton.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (isRecording) {
                // stop recording and release camera
                mMediaRecorder.stop();  // stop the recording
                releaseMediaRecorder(); // release the MediaRecorder object
                mCamera.lock();         // take camera access back from MediaRecorder

                // inform the user that recording has stopped
                setCaptureButtonText("Capture");
                isRecording = false;
            } else {
                // initialize video camera
                if (prepareVideoRecorder()) {
                    // Camera is available and unlocked, MediaRecorder is prepared,
                    // now you can start recording
                    mMediaRecorder.start();

                    // inform the user that recording has started
                    setCaptureButtonText("Stop");
                    isRecording = true;
                } else {
                    // prepare didn't work, release the camera
                    releaseMediaRecorder();
                    // inform user
                }
            }
        }
    }
);
```

> **Note:** In the above example, the `prepareVideoRecorder()` method refers to the example code shown in Configuring MediaRecorder. This method takes care of locking the camera, configuring and preparing the `MediaRecorder` instance.

Releasing the camera

Cameras are a resource that is shared by applications on a device. Your application can make use of the camera after getting an instance of `Camera`, and you must be particularly careful to release the camera object when your application stops using it, and as soon as your application is paused (`Activity.onPause()`). If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down.

To release an instance of the `Camera` object, use the `Camera.release()` method, as shown in the example code below.

```
public class CameraActivity extends Activity {
    private Camera mCamera;
    private SurfaceView mPreview;
    private MediaRecorder mMediaRecorder;

    ...

    @Override
    protected void onPause() {
        super.onPause();
        releaseMediaRecorder();       // if you are using MediaRecorder, release it first
        releaseCamera();              // release the camera immediately on pause event
    }

    private void releaseMediaRecorder(){
        if (mMediaRecorder != null) {
            mMediaRecorder.reset();   // clear recorder configuration
            mMediaRecorder.release(); // release the recorder object
            mMediaRecorder = null;
            mCamera.lock();           // lock camera for later use
        }
    }

    private void releaseCamera(){
        if (mCamera != null){
            mCamera.release();        // release the camera for other applications
            mCamera = null;
        }
    }
}
```

> **Caution:** If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down.

Saving media files

Media files created by users such as pictures and videos should be saved to a device's external storage directory (SD Card) to conserve system space and to allow users to access these files without their device. There are many possible directory locations to save media files on a device, however there are only two standard locations you should consider as a developer:

- `Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES)` - This method returns the standard, shared and recommended location for saving pictures and videos. This directory is shared (public), so other applications can easily discover, read, change and delete files saved in this location. If your application is uninstalled by the user, media files saved to this location will not be removed. To avoid interfering with users existing pictures and videos, you should create a sub-directory for your application's media files within this directory, as shown in the code sample below. This method is available in Android 2.2 (API Level 8), for equivalent calls in earlier API versions, see Saving Shared Files.
- `Context.getExternalFilesDir(Environment.DIRECTORY_PICTURES)` - This method returns a standard location for saving pictures and videos which are associated with your application. If your application is uninstalled, any files saved in this location are removed. Security is not enforced for files in this location and other applications may read, change and delete them.

The following example code demonstrates how to create a `File` or `Uri` location for a media file that can be used when invoking a device's camera with an `Intent` or as part of a Building a Camera App.

```
public static final int MEDIA_TYPE_IMAGE = 1;
public static final int MEDIA_TYPE_VIDEO = 2;

/** Create a file Uri for saving an image or video */
private static Uri getOutputMediaFileUri(int type){
      return Uri.fromFile(getOutputMediaFile(type));
}

/** Create a File for saving an image or video */
private static File getOutputMediaFile(int type){
    // To be safe, you should check that the SDCard is mounted
    // using Environment.getExternalStorageState() before doing this.

    File mediaStorageDir = new File(Environment.getExternalStoragePublicDirectory(
              Environment.DIRECTORY_PICTURES), "MyCameraApp");
    // This location works best if you want the created images to be shared
    // between applications and persist after your app has been uninstalled.

    // Create the storage directory if it does not exist
    if (! mediaStorageDir.exists()){
        if (! mediaStorageDir.mkdirs()){
            Log.d("MyCameraApp", "failed to create directory");
            return null;
        }
    }

    // Create a media file name
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    File mediaFile;
    if (type == MEDIA_TYPE_IMAGE){
        mediaFile = new File(mediaStorageDir.getPath() + File.separator +
        "IMG_"+ timeStamp + ".jpg");
    } else if(type == MEDIA_TYPE_VIDEO) {
        mediaFile = new File(mediaStorageDir.getPath() + File.separator +
        "VID_"+ timeStamp + ".mp4");
    } else {
        return null;
    }

    return mediaFile;
}
```

> **Note:** `Environment.getExternalStoragePublicDirectory()` is available in Android 2.2 (API Level 8) or higher. If you are targeting devices with earlier versions of Android, use `Environment.getExternalStorageDirectory()` instead. For more information, see Saving Shared Files.

To make the URI support work profiles, first convert the file URI to a content URI. Then, add the content URI to `EXTRA_OUTPUT` of an `Intent`.

For more information about saving files on an Android device, see Data Storage.

Camera features

Android supports a wide array of camera features you can control with your camera application, such as picture format, flash mode, focus settings, and many more. This section lists the common camera features, and briefly discusses how to use them. Most camera features can be accessed and set using the through `Camera.Parameters` object. However, there are several important features that require more than simple settings in `Camera.Parameters`. These features are covered in the following sections:

- Metering and focus areas
- Face detection
- Time lapse video

For general information about how to use features that are controlled through `Camera.Parameters`, review the Using camera features section. For more detailed information about how to use features controlled through the camera parameters object, follow the links in the feature list below to the API reference documentation.

Table 1. Common camera features sorted by the Android API Level in which they were introduced.

Feature	API Level	Description
Face Detection	14	Identify human faces within a picture and use them for focus, metering and white balance
Metering Areas	14	Specify one or more areas within an image for calculating white balance
Focus Areas	14	Set one or more areas within an image to use for focus
`White Balance Lock`	14	Stop or start automatic white balance adjustments
`Exposure Lock`	14	Stop or start automatic exposure adjustments
`Video Snapshot`	14	Take a picture while shooting video (frame grab)
Time Lapse Video	11	Record frames with set delays to record a time lapse video
`Multiple Cameras`	9	Support for more than one camera on a device, including front-facing and back-facing cameras
`Focus Distance`	9	Reports distances between the camera and objects that appear to be in focus
`Zoom`	8	Set image magnification
`Exposure Compensation`	8	Increase or decrease the light exposure level
`GPS Data`	5	Include or omit geographic location data with the image
`White Balance`	5	Set the white balance mode, which affects color values in the captured image
`Focus Mode`	5	Set how the camera focuses on a subject such as automatic, fixed, macro or infinity
`Scene Mode`	5	Apply a preset mode for specific types of photography situations such as night, beach, snow or candlelight scenes
`JPEG Quality`	5	Set the compression level for a JPEG image, which increases or decreases image output file quality and size
`Flash Mode`	5	Turn flash on, off, or use automatic setting
`Color Effects`	5	Apply a color effect to the captured image such as black and white, sepia tone or negative.
`Anti-Banding`	5	Reduces the effect of banding in color gradients due to JPEG compression
`Picture Format`	1	Specify the file format for the picture
`Picture Size`	1	Specify the pixel dimensions of the saved picture

> **Note:** These features are not supported on all devices due to hardware differences and software implementation. For information on checking the availability of features on the device where your application is running, see Checking feature availability.

Checking feature availability

The first thing to understand when setting out to use camera features on Android devices is that not all camera features are supported on all devices. In addition, devices that support a particular feature may support them to different levels or with different options. Therefore, part of your decision process as you develop a camera application is to decide what camera features you want to support and to what level. After making that decision, you should plan on including code in your camera application that checks to see if device hardware supports those features and fails gracefully if a feature is not available.

You can check the availability of camera features by getting an instance of a camera's parameters object, and checking the relevant methods. The following code sample shows you how to obtain a `Camera.Parameters` object and check if the camera supports the autofocus feature:

```
// get Camera parameters
Camera.Parameters params = mCamera.getParameters();

List<String> focusModes = params.getSupportedFocusModes();
if (focusModes.contains(Camera.Parameters.FOCUS_MODE_AUTO)) {
  // Autofocus mode is supported
}
```

You can use the technique shown above for most camera features. The `Camera.Parameters` object provides a `getSupported...()`, `is...Supported()` or `getMax...()` method to determine if (and to what extent) a feature is supported.

If your application requires certain camera features in order to function properly, you can require them through additions to your application manifest. When you declare the use of specific camera features, such as flash and auto-focus, Google Play restricts your application from being installed on devices which do not support these features. For a list of camera features that can be declared in your app manifest, see the manifest Features Reference.

Using camera features

Most camera features are activated and controlled using a `Camera.Parameters` object. You obtain this object by first getting an instance of the `Camera` object, calling the `getParameters()` method, changing the returned parameter object and then setting it back into the camera object, as demonstrated in the following example code:

```
// get Camera parameters
Camera.Parameters params = mCamera.getParameters();
// set the focus mode
params.setFocusMode(Camera.Parameters.FOCUS_MODE_AUTO);
// set Camera parameters
mCamera.setParameters(params);
```

This technique works for nearly all camera features, and most parameters can be changed at any time after you have obtained an instance of the `Camera` object. Changes to parameters are typically visible to the user immediately in the application's camera preview. On the software side, parameter changes may take several frames to actually take effect as the camera hardware processes the new instructions and then sends updated image data.

> **Important:** Some camera features cannot be changed at will. In particular, changing the size or orientation of the camera preview requires that you first stop the preview, change the preview size, and then restart the preview. Starting with Android 4.0 (API Level 14) preview orientation can be changed without restarting the preview.

Other camera features require more code in order to implement, including:

- Metering and focus areas
- Face detection
- Time lapse video

A quick outline of how to implement these features is provided in the following sections.

Metering and focus areas

In some photographic scenarios, automatic focusing and light metering may not produce the desired results. Starting with Android 4.0 (API Level 14), your camera application can provide additional controls to allow your app or users to specify areas in an image to use for determining focus or light level settings and pass these values to the camera hardware for use in capturing images or video.

Areas for metering and focus work very similarly to other camera features, in that you control them through methods in the `Camera.Parameters` object. The following code demonstrates setting two light metering areas for an instance of `Camera`:

```
// Create an instance of Camera
mCamera = getCameraInstance();

// set Camera parameters
Camera.Parameters params = mCamera.getParameters();

if (params.getMaxNumMeteringAreas() > 0){ // check that metering areas are supported
    List<Camera.Area> meteringAreas = new ArrayList<Camera.Area>();

    Rect areaRect1 = new Rect(-100, -100, 100, 100);    // specify an area in center of image
    meteringAreas.add(new Camera.Area(areaRect1, 600)); // set weight to 60%
    Rect areaRect2 = new Rect(800, -1000, 1000, -800);  // specify an area in upper right of image
    meteringAreas.add(new Camera.Area(areaRect2, 400)); // set weight to 40%
    params.setMeteringAreas(meteringAreas);
}

mCamera.setParameters(params);
```

The `Camera.Area` object contains two data parameters: A `Rect` object for specifying an area within the camera's field of view and a weight value, which tells the camera what level of importance this area should be given in light metering or focus calculations.

The `Rect` field in a `Camera.Area` object describes a rectangular shape mapped on a 2000 x 2000 unit grid. The coordinates -1000, -1000 represent the top, left corner of the camera image, and coordinates 1000, 1000 represent the bottom, right corner of the camera image, as shown in the illustration below.

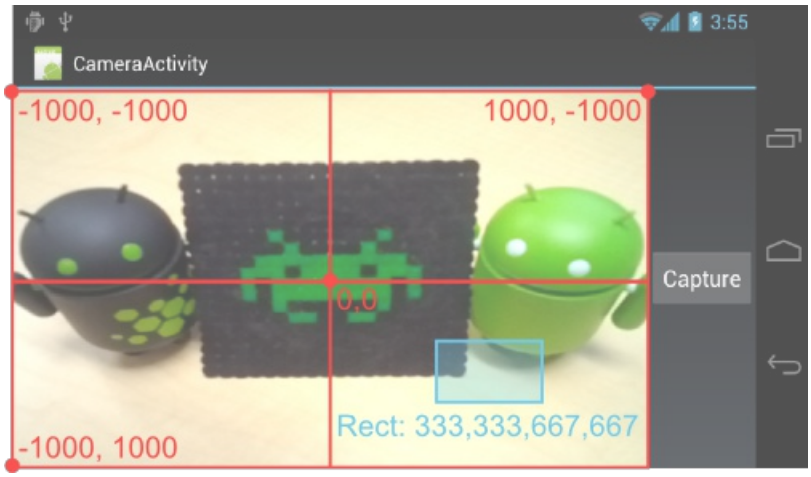

Figure 1. The red lines illustrate the coordinate system for specifying a `Camera.Area` within a camera preview. The blue box shows the location and shape of an camera area with the `Rect` values 333,333,667,667.

The bounds of this coordinate system always correspond to the outer edge of the image visible in the camera preview and do not shrink or expand with the zoom level. Similarly, rotation of the image preview using `Camera.setDisplayOrientation()` does not remap the coordinate system.

Face detection

For pictures that include people, faces are usually the most important part of the picture, and should be used for determining both focus and white balance when capturing an image. The Android 4.0 (API Level 14) framework provides APIs for identifying faces and calculating picture settings using face recognition technology.

> **Note:** While the face detection feature is running, `setWhiteBalance(String)`, `setFocusAreas(List<Camera.Area>)` and `setMeteringAreas(List<Camera.Area>)` have no effect.

Using the face detection feature in your camera application requires a few general steps:

- Check that face detection is supported on the device
- Create a face detection listener
- Add the face detection listener to your camera object
- Start face detection after preview (and after *every* preview restart)

The face detection feature is not supported on all devices. You can check that this feature is supported by calling `getMaxNumDetectedFaces()`. An example of this check is shown in the `startFaceDetection()` sample method below.

In order to be notified and respond to the detection of a face, your camera application must set a listener for face detection events. In order to do this, you must create a listener class that implements the `Camera.FaceDetectionListener` interface as shown in the example code below.

```
class MyFaceDetectionListener implements Camera.FaceDetectionListener {

    @Override
    public void onFaceDetection(Face[] faces, Camera camera) {
        if (faces.length > 0){
            Log.d("FaceDetection", "face detected: "+ faces.length +
                    " Face 1 Location X: " + faces[0].rect.centerX() +
                    "Y: " + faces[0].rect.centerY() );
        }
    }
}
```

After creating this class, you then set it into your application's `Camera` object, as shown in the example code below:

```
mCamera.setFaceDetectionListener(new MyFaceDetectionListener());
```

Your application must start the face detection function each time you start (or restart) the camera preview. Create a method for starting face detection so you can call it as needed, as shown in the example code below.

```
public void startFaceDetection(){
    // Try starting Face Detection
    Camera.Parameters params = mCamera.getParameters();

    // start face detection only *after* preview has started
    if (params.getMaxNumDetectedFaces() > 0){
        // camera supports face detection, so can start it:
        mCamera.startFaceDetection();
    }
}
```

You must start face detection *each time* you start (or restart) the camera preview. If you use the preview class shown in Creating a preview class, add your `startFaceDetection()` method to both the `surfaceCreated()` and `surfaceChanged()` methods in your preview class, as shown in the sample code below.

```
public void surfaceCreated(SurfaceHolder holder) {
    try {
        mCamera.setPreviewDisplay(holder);
        mCamera.startPreview();

        startFaceDetection(); // start face detection feature

    } catch (IOException e) {
        Log.d(TAG, "Error setting camera preview: " + e.getMessage());
    }
}

public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {

    if (mHolder.getSurface() == null){
        // preview surface does not exist
        Log.d(TAG, "mHolder.getSurface() == null");
        return;
    }

    try {
        mCamera.stopPreview();

    } catch (Exception e){
        // ignore: tried to stop a non-existent preview
        Log.d(TAG, "Error stopping camera preview: " + e.getMessage());
    }

    try {
        mCamera.setPreviewDisplay(mHolder);
        mCamera.startPreview();

        startFaceDetection(); // re-start face detection feature

    } catch (Exception e){
        // ignore: tried to stop a non-existent preview
        Log.d(TAG, "Error starting camera preview: " + e.getMessage());
    }
}
```

Note: Remember to call this method *after* calling `startPreview()`. Do not attempt to start face detection in the `onCreate()` method of your camera app's main activity, as the preview is not available by this point in your application's the execution.

Time lapse video

Time lapse video allows users to create video clips that combine pictures taken a few seconds or minutes apart. This feature uses `MediaRecorder` to record the images for a time lapse sequence.

To record a time lapse video with `MediaRecorder`, you must configure the recorder object as if you are recording a normal video, setting the captured frames per second to a low number and using one of the time lapse quality settings, as shown in the code example below.

```
// Step 3: Set a CamcorderProfile (requires API Level 8 or higher)
mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_TIME_LAPSE_HIGH));
...
// Step 5.5: Set the video capture rate to a low number
mMediaRecorder.setCaptureRate(0.1); // capture a frame every 10 seconds
```

These settings must be done as part of a larger configuration procedure for `MediaRecorder`. For a full configuration code example, see Configuring MediaRecorder. Once the configuration is complete, you start the video recording as if you were recording a normal video clip. For more information about configuring and running `MediaRecorder`, see Capturing videos.