Detect common gestures

A "touch gesture" occurs when a user places one or more fingers on the touch screen, and your application interprets that pattern of touches as a particular gesture. There are correspondingly two phases to gesture detection:

1. Gather data about touch events.
2. Interpret the data to see if it meets the criteria for any of the gestures your app supports.

Refer to the following related resources:

- Input Events API Guide
- Sensors Overview
- Making the View Interactive

Support library classes

The examples in this lesson use the GestureDetectorCompat and MotionEventCompat classes. These classes are in the Support Library. You should use Support Library classes where possible to provide compatibility with devices running Android 1.6 and higher. Note that MotionEventCompat is *not* a replacement for the MotionEvent class. Rather, it provides static utility methods to which you pass your MotionEvent object in order to receive the desired action associated with that event.

Gather data

When a user places one or more fingers on the screen, this triggers the callback onTouchEvent() on the View that received the touch events. For each sequence of touch events (position, pressure, size, addition of another finger, etc.) that is ultimately identified as a gesture, onTouchEvent() is fired several times.

The gesture starts when the user first touches the screen, continues as the system tracks the position of the user's finger(s), and ends by capturing the final event of the user's fingers leaving the screen. Throughout this interaction, the MotionEvent delivered to onTouchEvent() provides the details of every

interaction. Your app can use the data provided by the MotionEvent to determine if a gesture it cares about happened.

Capture touch events for an Activity or View

To intercept touch events in an Activity or View, override the onTouchEvent() callback.

The following snippet uses getActionMasked() to extract the action the user performed from the event parameter. This gives you the raw data you need to determine if a gesture you care about occurred:

KotlinJava

```
public class MainActivity extends Activity {
...
// This example shows an Activity, but you would use the same approach if
// you were subclassing a View.
@Override
public boolean onTouchEvent(MotionEvent event){

    int action = MotionEventCompat.getActionMasked(event);

    switch(action) {
        case (MotionEvent.ACTION_DOWN) :
            Log.d(DEBUG_TAG,"Action was DOWN");
            return true;
        case (MotionEvent.ACTION_MOVE) :
            Log.d(DEBUG_TAG,"Action was MOVE");
            return true;
        case (MotionEvent.ACTION_UP) :
            Log.d(DEBUG_TAG,"Action was UP");
            return true;
        case (MotionEvent.ACTION_CANCEL) :
            Log.d(DEBUG_TAG,"Action was CANCEL");
            return true;
        case (MotionEvent.ACTION_OUTSIDE) :
            Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                    "of current screen element");
            return true;
        default :
            return super.onTouchEvent(event);
    }
}
```

You can then do your own processing on these events to determine if a gesture occurred. This is the kind of processing you would have to do for a custom gesture. However, if your app uses common gestures such as double tap, long press, fling, and so on, you can take advantage of the GestureDetector class. GestureDetector makes it easy for you to detect common gestures without processing the individual touch events yourself. This is discussed below in Detect Gestures.

Capture touch events for a single view

As an alternative to onTouchEvent(), you can attach an View.OnTouchListener object to any View object using the setOnTouchListener() method. This makes it possible to to listen for touch events without subclassing an existing View. For example:

KotlinJava

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
    public boolean onTouch(View v, MotionEvent event) {
        // ... Respond to touch events
        return true;
    }
});
```

Beware of creating a listener that returns false for the ACTION_DOWN event. If you do this, the listener will not be called for the subsequent ACTION_MOVE and ACTION_UP string of events. This is because ACTION_DOWN is the starting point for all touch events.

If you are creating a custom View, you can override onTouchEvent(), as described above.

Detect gestures

Android provides the GestureDetector class for detecting common gestures. Some of the gestures it supports include onDown(), onLongPress(), onFling(), and so on. You can use GestureDetector in conjunction with the onTouchEvent() method described above.

Detect all supported gestures

When you instantiate a GestureDetectorCompat object, one of the parameters it takes is a class that implements the GestureDetector.OnGestureListener interface. GestureDetector.OnGestureListener not

ifies users when a particular touch event has occurred. To make it possible for your GestureDetector object to receive events, you override the View or Activity's onTouchEvent() method, and pass along all observed events to the detector instance.

In the following snippet, a return value of `true` from the individual `on<TouchEvent>` methods indicates that you have handled the touch event. A return value of `false` passes events down through the view stack until the touch has been successfully handled.

Run the following snippet to get a feel for how actions are triggered when you interact with the touch screen, and what the contents of the MotionEvent are for each touch event. You will realize how much data is being generated for even simple interactions.

KotlinJava

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG = "Gestures";
    private GestureDetectorCompat mDetector;

    // Called when the activity is first created.
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Instantiate the gesture detector with the
        // application context and an implementation of
        // GestureDetector.OnGestureListener
        mDetector = new GestureDetectorCompat(this,this);
        // Set the gesture detector as the double tap
        // listener.
        mDetector.setOnDoubleTapListener(this);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        if (this.mDetector.onTouchEvent(event)) {
            return true;
        }
        return super.onTouchEvent(event);
    }
```

```
@Override
public boolean onDown(MotionEvent event) {
    Log.d(DEBUG_TAG,"onDown: " + event.toString());
    return true;
}

@Override
public boolean onFling(MotionEvent event1, MotionEvent event2,
        float velocityX, float velocityY) {
    Log.d(DEBUG_TAG, "onFling: " + event1.toString() + event2.toString());
    return true;
}

@Override
public void onLongPress(MotionEvent event) {
    Log.d(DEBUG_TAG, "onLongPress: " + event.toString());
}

@Override
public boolean onScroll(MotionEvent event1, MotionEvent event2, float distanceX,
        float distanceY) {
    Log.d(DEBUG_TAG, "onScroll: " + event1.toString() + event2.toString());
    return true;
}

@Override
public void onShowPress(MotionEvent event) {
    Log.d(DEBUG_TAG, "onShowPress: " + event.toString());
}

@Override
public boolean onSingleTapUp(MotionEvent event) {
    Log.d(DEBUG_TAG, "onSingleTapUp: " + event.toString());
    return true;
}

@Override
public boolean onDoubleTap(MotionEvent event) {
    Log.d(DEBUG_TAG, "onDoubleTap: " + event.toString());
    return true;
}

@Override
public boolean onDoubleTapEvent(MotionEvent event) {
```

```
        Log.d(DEBUG_TAG, "onDoubleTapEvent: " + event.toString());
        return true;
    }

    @Override
    public boolean onSingleTapConfirmed(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapConfirmed: " + event.toString());
        return true;
    }
}
```

Detect a subset of supported gestures

If you only want to process a few gestures, you can extend GestureDetector.SimpleOnGestureListener instead of implementing the GestureDetector.OnGestureListener interface.

GestureDetector.SimpleOnGestureListener provides an implementation for all of the on<*TouchEvent*> methods by returning false for all of them. Thus you can override only the methods you care about. For example, the snippet below creates a class that extends GestureDetector.SimpleOnGestureListener and overrides onFling() and onDown().

Whether or not you use GestureDetector.OnGestureListener, it's best practice to implement an onDown() method that returns true. This is because all gestures begin with an onDown() message. If you return false from onDown(), as GestureDetector.SimpleOnGestureListener does by default, the system assumes that you want to ignore the rest of the gesture, and the other methods of GestureDetector.OnGestureListener never get called. This has the potential to cause unexpected problems in your app. The only time you should return false from onDown() is if you truly want to ignore an entire gesture.

Kotlin Java

```
public class MainActivity extends Activity {

    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new MyGestureListener());
    }
```

```
    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    class MyGestureListener extends GestureDetector.SimpleOnGestureListener {
        private static final String DEBUG_TAG = "Gestures";

        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(DEBUG_TAG,"onDown: " + event.toString());
            return true;
        }

        @Override
        public boolean onFling(MotionEvent event1, MotionEvent event2,
                float velocityX, float velocityY) {
            Log.d(DEBUG_TAG, "onFling: " + event1.toString() + event2.toString());
            return true;
        }
    }
}
```

Track touch and pointer movements

This lesson describes how to track movement in touch events.

A new onTouchEvent() is triggered with an ACTION_MOVE event whenever the current touch contact position, pressure, or size changes. As described in Detecting Common Gestures, all of these events are recorded in the MotionEvent parameter of onTouchEvent().

Because finger-based touch isn't always the most precise form of interaction, detecting touch events is often based more on movement than on simple contact. To help apps distinguish between movement-based gestures (such as a swipe) and non-movement gestures (such as a single tap), Android includes the notion of "touch slop". Touch slop refers to the distance in pixels a user's touch can wander before the gesture is interpreted as a movement-

based gesture. For more discussion of this topic, see Managing Touch Events in a ViewGroup.

There are several different ways to track movement in a gesture, depending on the needs of your application. For example:

- The starting and ending position of a pointer (for example, move an on-screen object from point A to point B).
- The direction the pointer is traveling in, as determined by the x and y coordinates.
- History. You can find the size of a gesture's history by calling the MotionEvent method getHistorySize(). You can then obtain the positions, sizes, time, and pressures of each of the historical events by using the motion event's getHistorical<*Value*> methods. History is useful when rendering a trail of the user's finger, such as for touch drawing. See the MotionEvent reference for details.
- The velocity of the pointer as it moves across the touch screen.

Refer to the following related resources:

- Input Events API Guide
- Sensors Overview
- Making the View Interactive

Track velocity

You could have a movement-based gesture that is simply based on the distance and/or direction the pointer traveled. But velocity often is a determining factor in tracking a gesture's characteristics or even deciding whether the gesture occurred. To make velocity calculation easier, Android provides the VelocityTracker class. VelocityTracker helps you track the velocity of touch events. This is useful for gestures in which velocity is part of the criteria for the gesture, such as a fling.

Here is a simple example that illustrates the purpose of the methods in the VelocityTracker API:

Kotlin Java

```
public class MainActivity extends Activity {
    private static final String DEBUG_TAG = "Velocity";
        ...
    private VelocityTracker mVelocityTracker = null;
```

```
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        int index = event.getActionIndex();
        int action = event.getActionMasked();
        int pointerId = event.getPointerId(index);

        switch(action) {
            case MotionEvent.ACTION_DOWN:
                if(mVelocityTracker == null) {
                    // Retrieve a new VelocityTracker object to watch the
                    // velocity of a motion.
                    mVelocityTracker = VelocityTracker.obtain();
                }
                else {
                    // Reset the velocity tracker back to its initial state.
                    mVelocityTracker.clear();
                }
                // Add a user's movement to the tracker.
                mVelocityTracker.addMovement(event);
                break;
            case MotionEvent.ACTION_MOVE:
                mVelocityTracker.addMovement(event);
                // When you want to determine the velocity, call
                // computeCurrentVelocity(). Then call getXVelocity()
                // and getYVelocity() to retrieve the velocity for each pointer ID.
                mVelocityTracker.computeCurrentVelocity(1000);
                // Log velocity of pixels per second
                // Best practice to use VelocityTrackerCompat where possible.
                Log.d("", "X velocity: " + mVelocityTracker.getXVelocity(pointerId));
                Log.d("", "Y velocity: " + mVelocityTracker.getYVelocity(pointerId));
                break;
            case MotionEvent.ACTION_UP:
            case MotionEvent.ACTION_CANCEL:
                // Return a VelocityTracker object back to be re-used by others.
                mVelocityTracker.recycle();
                break;
        }
        return true;
    }
}
```

Note: You should calculate velocity after an `ACTION_MOVE` event, not after `ACTION_UP`. After an `ACTION_UP`, the X and Y velocities will be 0.

Use pointer capture

Some apps, such as games, remote desktop, and virtualization clients, greatly benefit from getting control over the mouse pointer. Pointer capture is a feature available in Android 8.0 (API level 26) and later that provides such control by delivering all mouse events to a focused view in your app.

Request pointer capture

A view in your app can request pointer capture only when the view hierarchy that contains it has focus. For this reason, you should request pointer capture when there's a specific user action on the view, such as during an onClick() event, or in the onWindowFocusChanged() event handler of your activity.

To request pointer capture, call the requestPointerCapture() method on the view. The following code example shows how to request pointer capture when the user clicks a view:

KotlinJava

```
@Override
public void onClick(View view) {
    view.requestPointerCapture();
}
```

Once the request to capture the pointer is successful, Android calls onPointerCaptureChange(true). The system delivers the mouse events to the focused view in your app as long as it's in the same view hierarchy as the view that requested the capture. Other apps stop receiving mouse events until the capture is released, including ACTION_OUTSIDE events. Android delivers pointer events from sources other than the mouse normally, but the mouse pointer is not visible anymore.

Handle captured pointer events

Once a view has successfully acquired the pointer capture, Android starts delivering the mouse events. Your focused view can handle the events by performing one of the following tasks:

1. If you're using a custom view, override onCapturedPointerEvent(MotionEvent).
2. Otherwise, register an OnCapturedPointerListener.

The following code example shows how to implement onCapturedPointerEvent(MotionEvent):

Kotlin Java

```
@Override
public boolean onCapturedPointerEvent(MotionEvent motionEvent) {
  // Get the coordinates required by your app
  float verticalOffset = motionEvent.getY();
  // Use the coordinates to update your view and return true if the event was
  // successfully processed
  return true;
}
```

The following code example shows how to register an OnCapturedPointerListener:

Kotlin Java

```
myView.setOnCapturedPointerListener(new View.OnCapturedPointerListener() {
  @Override
  public boolean onCapturedPointer (View view, MotionEvent motionEvent) {
    // Get the coordinates required by your app
    float horizontalOffset = motionEvent.getX();
    // Use the coordinates to update your view and return true if the event was
    // successfully processed
    return true;
  }
});
```

Whether you use a custom view or register a listener, your view receives a MotionEvent with pointer coordinates that specify relative movements such as X/Y deltas, similar to the coordinates delivered by a trackball device. You can retrieve the coordinates by using getX() and getY().

Release pointer capture

The view in your app can release the pointer capture by calling releasePointerCapture(), as shown in the following code example:

```
@Override
public void onClick(View view) {
    view.releasePointerCapture();
}
```