Kivy Cookbook

Enhance your skills in developing multitouch applications with Kivy

Hugo Solis
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Input, Motion, and Touch'
- A synopsis of the book’s content
- More information on Kivy Cookbook
Hugo Solis is an assistant professor in the physics department at the University of Costa Rica. In the same institution, he collaborates with CITIC and CICIMA. His current research interests include computational cosmology, complexity, and the influence of hydrogen on material properties. He has wide experience in languages, including C/C++ and Python for scientific programming and visualization. He is a member of the Free Software Foundation and has contributed code to some free software projects. He has also been a technical reviewer for *Mastering Object-oriented Python*, *Kivy: Interactive Applications in Python*, and *Learning Object-Oriented Programming* by Packt Publishing. Currently, he is in charge of the IFT, a Costa Rican scientific nonprofit organization for the multidisciplinary practice of physics (http://iftucr.org).
Preface

The book is full of practical task-based recipes to be performed while developing multitouch applications with Kivy. The focus of this book is to guide you through the Kivy framework to develop apps and get your apps ready for distribution in App Stores and Android devices. We'll focus on common, real-world scenarios. You'll be able to leverage these recipes right away, which allows you to create most diverse apps and learn how to distribute them using the Kivy framework, and widen your proficiency in developing multitouch applications with Kivy.

What this book covers

Chapter 1, *Kivy and the Kv Language*, introduces the Python code, Kv language, widgets, and Kivy garden. The topics covered in this chapter lay the foundation for the remaining code samples in each chapter.

Chapter 2, *Input, Motion, and Touch*, demonstrates some of the most common recipient-related management tasks, such as evolving to the touchscreen, using the gyroscope, and detecting multitapping. You'll also learn how to recognize touch shapes, record gestures, and differentiate between touch and motion events.

Chapter 3, *Events*, shows how to schedule a one-time event and a repetitive event. Triggering events, defining widget events, creating custom events are also covered in this chapter.

Chapter 4, *Widgets*, covers the manipulation of widget tree, traversing of tree. It also covers the FloatLayout, BoxLayout, StackLayout, AnchorLayout, and action bar.

Chapter 5, *Graphics – Canvas and Instructions*, explains various methods, such as separating with the screen manager, using context instructions, and using drawing instructions. You'll learn how to modify with multitouching and storing and retrieving the coordinate space context.

Chapter 6, *Advancing Graphics – Shaders and Rendering*, covers Carousel, layouts, and shaders. The other topics covered in this chapter are rendering in a Framebuffer and optimizing graphics.
Chapter 7, *The API in Detail*, introduces API and covers the logging objects, parsing, applying utils, using spelling, adding effects, and adding text.

Chapter 8, *Packaging our Apps for PC*, shows how to perform packing for Windows, Linux, and MacOS. It also shows multimedia for Windows and running apps in Windows and MacOS.

Chapter 9, *Kivy for Mobile Devices*, explains various methods, such as packaging for iOS and resizing the screen into iOS. You'll learn preparing for the App Store, packaging for Android, and resizing the screen into Android.
Input, Motion, and Touch

In this chapter, we will cover:

- Using the mouse
- Evolving to the touchscreen
- Working with the accelerometer
- Using the gyroscope
- The Differences between the touch and motion events
- Recognizing touch shapes
- Detecting multitapping
- Grabbing touch events
- Recording gestures

Introduction

The Kivy framework is able to handle most common types of input and actions such as mouse, touchscreen, accelerometer, and gyroscope that will be reviewed in this chapter. It handles the native multitouch protocols on the following platforms: Tuio, WM_Touch, MacMultitouchSupport, MT Protocol A/B, and Android.

The class of all input events is motion event. It generates two kinds of events; one of them is touch events—a motion event that contains at least an x and y position. All the touch events are dispatched across the widget tree. The no-touch events are the rest. The accelerometer is a continuous event, without position. It never starts or stops. These events are not dispatched across the widget tree.
Using the mouse

This recipe will teach you how to use the first kind of input, probably the most used, the mouse. We will consider the mouse input as a touch event, so the \((x, y)\) position of the touch (0-1 range) will be scaled to the window size (width/height) and dispatched to the following:

- **on_touch_down**: An event is fired when a touch down event is initiated. Subsequently, the respective widget's `on_touch_down()` method is called.
- **on_touch_move**: An event is fired when a touch event moves (changes location). Subsequently, the respective widget's `on_touch_move()` method is called.
- **on_touch_up**: An event is fired when a down event is released (terminated). Subsequently, the respective widget's `on_touch_up()` method is called.

Getting ready

In this recipe, we will use the Kv language for the design of the widgets, so you will need to be familiar with the language or have completed the previous chapter. Also, this recipe will use the common Button widget for reference.

How to do it...

Follow the steps in this recipe:

1. First, let's start with the KV file:

   ```kivy
   <MyW>:
       Button:
           id: button1
           text: 'Hello'
   ```

   In the class of the widget in the Python code, we need to override the method `on_touch_down` because the method is already defined in the Kivy framework.

2. Change the button text with the information in `touch.button`:

   ```python
   import kivy
   kivy.require('1.9.0')
   from kivy.app import App
   from kivy.uix.widget import Widget

   class MyW(Widget):
       
       def on_touch_down(self, touch):
           if 'button' in touch.profile:
               ```
self.ids.button1.text = touch.button

class elApp(App):
    
    def build(self):
        return MyW()

    if __name__ == '__main__':
        elApp().run()

How it works...

Let's start with the KV file. The first line is the name of the rule, similar to the class name in the Python file. The second line is the definition of the button widget, and the third line is the ID of the button, which is necessary to do the call of the widget inside the Python code, and the fourth line is the definition of the text in the button.

With regard to the Python code, the initial four lines are usual to use Kivy and the button widget. Note the following fifth line:

    class MyW(Widget):

This is where we define the class associated with our rule in the KV file. Now consider the sixth line:

    def on_touch_down(self, touch):

Here, we start the declaration of the dispatched on_touch_down method; you must note the parameter touch in the declaration, and this parameter is necessary to call the event using the input—in this case—the mouse. Next, the seventh line is:

    if 'button' in touch.profile:

This is a verification line, because we need to be sure that the input device used in the platform, where we are running our code, supports the button profile. If this line is not present when the next line is performed, the app could crash. Finally, the eighth line:

    self.ids.button1.text = touch.button

This line is where we do the call to the button profile, which gives the information on which mouse button is touched by the user (the right one, left one, scroll up, and so on). Using this information, we change the text in our button that has ID button1. The last lines of the Python code are usual to run and display our Kivy interface. Just remember that the initial part of the name of the class is as follows:

    class elApp(App):
Note that it will be the same as the KV file, so the name of the KV file in this case is e1.kv.

A last thing to remark is that we do not limit the input event to the button, so the touch can occur anywhere within the window.

**There's more...**

Another interesting profile that we could call for the mouse is the `pos` attribute, which provides us with the position vector of the click in the window. Let's change the seventh line of the Python code to:

```
self.ids.button1.text = str(touch.pos)
```

When we are calling the `pos` profile and changing the text in the button widget to the position where the click is performed, the `str()` built-in object is important because the `touch.pos` return is a vector. This means that we need to convert to a string to avoid compatibility issues.

**See also**

If you want to run your interface, take a look at our recipe *Running your code*. To get further details about widgets, see the recipes in *Chapter 4, Widgets*.

**Evolving to the touchscreen**

In this recipe, we are evolving to the touchscreen. Here, you will see the basic differences between the mouse and the touchscreen. This will give us more options than with the mouse device.

**Getting ready**

For this recipe, we will use the Kv language for the design of the widgets, so make sure that you are confident with it and refresh your knowledge if necessary. Also, this recipe will use the common button and label widgets for reference. Obviously, to get more benefit, a touchscreen device is useful to run the code.

**How to do it...**

Perform the following steps:

1. In the KV file, declare the button and the label:

   ```
   <MyW>:
     Button:
       id: button1
   ```
2. In the class of the widget in the Python code, we need to override the method `on_touch_down`
3. Change the button text with the information in `touch.button`
4. Change the label text with the information in `touch.pressure`

```python
import kivy
kivy.require('1.9.0')

from kivy.app import App
from kivy.uix.widget import Widget

class MyW(Widget):
    def on_touch_down(self, touch):
        if 'button' in touch.profile:
            self.ids.button1.text = touch.button

        if 'pressure' in touch.profile:
            self.ids.label1.text = str(touch.pressure)

class e2App(App):
    def build(self):
        return MyW()

if __name__ == '__main__':
e2App().run()
```

**How it works...**

Well, let's first review the KV file. The first line is the name of the rule, the same as of the class in the Python file. The second line is the definition of the button widget, the third line is the ID of the button, which is necessary to do the call of the widget inside the Python code, and the fourth line is the definition of the text in the button. The fifth line is the definition of the label widget. The sixth line is where we give the ID to the label, the seventh line is where we give the position to the label. I should point out that the button is using the default position (0, 0), so we will have to give a different position to the label widget to avoid overlapping. The eight is the definition of the initial text in the label.
With regard to the Python code, the initial four lines are usual to use Kivy and the widgets. Note the fifth one:

```python
class MyW(Widget):
```
This is where we define the class associated with our rule in the KV file. The sixth line:

```python
def on_touch_down(self, touch):
```
Here, we start the declaration of the dispatched `on_touch_down` method. You must note the parameter `touch` in the declaration as this parameter is necessary to call the event, which is done by the input (in this case, the mouse). The seventh line is:

```python
if 'button' in touch.profile:
```
It is a verification line because we need to be sure that the input device used in the platform, where we are running our code, supports the button profile. If this line is not present when the next one is performed, the app could crash. The eighth line is:

```python
self.ids.button1.text = touch.button
```
This line is where we do the call to the profile button, which gives the information of what button is touched by the user (the right, left, scroll up button, and so on) and we changed the text in the button with the ID `button1` with that information. The ninth line is:

```python
if 'pressure' in touch.profile:
```
This is the other verification line and is important because, due to the fact that pressure is specific to some touchscreen devices, it is necessary to avoid crashes. The tenth and eleventh line:

```python
self.ids.label1.text = str(touch.pressure)
```
Those lines are where we change the text of the label to the pressure value of the specific touch.

The last lines of the Python code are usual to run and display our Kivy interface. Just remember that the initial part of the name of the class is:

```python
class e2App(App):
```
It will be the same in relation to the KV file. For example, the name of the KV file in this case is `e2.kv`.

The final thing to remark is that we do not limit the input event to the button, so the touch can occur anywhere within the window or the screen.
Actually, your device could have more profiles than we saw in this recipe. Let's change the code to know that for your specific device. Remove the ninth line and change the tenth and eleventh lines to:

```
self.ids.label1.text =
str(touch.profile)
```

These lines will change the label's text for all the profiles available for your device. Thus, you can get more yield of your app.

### See also

If you want to run your interface, take a look at our recipe Running your code, and to get more detail about widgets, see the recipes in *Chapter 4, Widgets*. If you want to run the code in a mobile device, see *Chapter 9, Kivy for Mobile Devices*.

### Working with the accelerometer

Nowadays, it is common that mobile devices are equipped with an accelerometer. Thereby, it is relevant to consider this input to develop fancy apps. The accelerometer is one kind of event that is considered as a no-touch event because it has neither start nor an end, it is always providing information.

### Getting ready

In this recipe, we will use the Kv language for the design of the widgets, so, again, you will need to be familiar with the Kv language. Also, this recipe will use the common button and label widgets for reference. Obviously, to get more benefit, a device with an accelerometer is useful to run the code.

It is necessary to install a package for the use of the accelerometer in any mobile device; there is a way to deal with every specific device, but we do not have to reinvent the wheel. Let's use a Plyer package, which is an effort from many of the Kivy developers. To install the package from the shell, use:

```
$ sudo pip plyer install
```

Here where we are using pip to install the Plyer package, if you do not have pip installed in your computer, see the recipe *Using Kivy Garden in Chapter 1, Kivy and the Kv language*. 
How to do it...

To complete this recipe, follow these steps:

1. In the KV file, define the four labels and the button:
   
   ```latex
   <Accel>:
   Label:
   id: label1
   pos: 150, 300
   text: 'X: '

   Label:
   id: label2
   pos: 150, 250
   text: 'Y: '

   Label:
   id: label3
   pos: 150, 200
   text: 'Z: '

   Label:
   id: status
   pos: 150, 150
   text: ''

   Button:
   id: button1
   pos: 150, 50
   text: 'Start'
   on_press: root.pressed1()
   ```

2. In the Python file, import the usual packages to use Kivys.
3. Import the Clock and Plyer packages.
4. In the class for the rule, define a class to standby the accelerometer.
5. Define the method to start to retrieve data from the accelerometer.
6. Define the method when the button is pressed:

   ```python
   from kivy.app import App
   from kivy.properties import ObjectProperty
   from kivy.uix.widget import Widget
   ```
from kivy.clock import Clock
from plyer import accelerometer

class Accel(Widget):
    def __init__(self):
        super(Accel, self).__init__()
        self.sensorEnabled = False

    def get_acceleration(self, dt):
        val = accelerometer.acceleration
        self.ids.label1.text = "X: " + str(val[0])
        self.ids.label2.text = "Y: " + str(val[1])
        self.ids.label3.text = "Z: " + str(val[2])

    def pressed1(self):
        try:
            if not self.sensorEnabled:
                accelerometer.enable()
                Clock.schedule_interval(self.get_acceleration,
                1 / 20.)
                self.sensorEnabled = True
                self.ids.button1.text = "Stop"
            else:
                accelerometer.disable()
                Clock.unschedule(self.get_acceleration)
                self.sensorEnabled = False
                self.ids.button1.text = "Start"
        except NotImplementedError:
            import traceback; traceback.print_exc()
            self.ids.status.text = "Accelerometer is not supported for your platform"


class e3App(App):
    def build(self):
        return Accel()

if __name__ == '__main__':
e3App().run()
How it works...

First, review the KV file. Note that the first line is the name of the rule, similar to the class in the Python file. Included in the next lines are the definitions of the four labels that we will use, so note the text of the label with the ID status. It will likely be blank, but it will be visible only when we modify this field from the Python code.

In the Python code, the first three lines are usual to import kivy and the widgets. The fourth line:

```python
from kivy.clock import Clock
```

This will import the clock, which is used to get the accelerometer value in a time lap because, as we already said, this input is continuously providing data. The fifth line:

```python
from plyer import accelerometer
```

This is importing the package plyer, which is the one that makes it possible to retrieve the data from an iOS or Android device. The sixth line:

```python
class Accel(Widget):
```

This is the line where the class for the rule starts. Its name must be the same as the Kivy rule. The seventh, eighth, and ninth lines are:

```python
def __init__(self):
    super(Accel, self).__init__()
    self.sensorEnabled = False
```

These define the method that initializes the class and put in standby the accelerometer to be used later. The tenth, eleventh, twelfth, thirteenth, and fourteenth lines are:

```python
def get_acceleration(self, dt):
    val = accelerometer.acceleration
    self.ids.label1.text = "X: " + str(val[0])
    self.ids.label2.text = "Y: " + str(val[1])
    self.ids.label3.text = "Z: " + str(val[2])
```

These lines are used to retrieve the data of the accelerometer and will modify the text of the first three labels with the data of the acceleration. Now, look at the fifteenth line:

```python
def pressed1(self):
```

This is where we define the method performed when the button is pressed. In the sixteenth line, we start a try sentence to start getting data from the accelerometer. In the seventeenth line, it is shown that if the accelerometer is disabled, enable it or vice versa. The eighteenth line is:

```python
Clock.schedule_interval(self.get_acceleration, 1 / 20.)
```
In this line, we do the acquisition of the acceleration in time intervals. The acceleration is acquired by calling the `get_acceleration` method. The next class is usual to display our Kivy interface.

**There's more...**

Also, the `plyer` package allows you to interact with the compass and the GPS of the mobile device using a similar procedure. In the case of the compass, it is very easy to change. The change in the code is the word `gyroscope` for `compass` and `acceleration` for `orientation`.

**See also**

To get more details about widgets, see the recipes in Chapter 4, *Widgets*, and if you want to run the code in a mobile device, see Chapter 9, *Kivy for Mobile Devices*.

**Using the gyroscope**

Mobile devices today also have a gyroscope. The gyroscope is considered as a no-touch event. This kind of event does not have a start or an end; it is always providing information.

**Getting ready**

For this recipe, we will use the Kv language for the design of the widgets. Also, this recipe will use the common button and label widgets for reference. Obviously, to run the code of this recipe properly, a device with a gyroscope is necessary.

It is necessary to install a package to use the gyroscope with any mobile device. While there is a way to deal with specific devices, we do not have to reinvent the wheel. Let's use the `plyer` package, which is an effort from many of the Kivy developers. To install the package from the shell, use:

`$ sudo pip plyer install`

**How to do it...**

To complete this recipe, follow these steps:

1. In the KV file, define the four labels and the button:
   ```xml
   <Acce>:
       Label:
           id: labell
           pos: 150, 300
   ```
2. In the Python file, import the usual packages to use Kivy.

3. Import the Clock and plyer packages.

4. In the class for the rule, define a class to stand by the gyroscope.

5. Define the method to start to retrieve data from the gyroscope.

6. Define the method when the button is pressed:

```python
from kivy.app import App
from kivy.properties import ObjectProperty
from kivy.uix.widget import Widget
from kivy.clock import Clock
from plyer import gyroscope

class Acce(Widget):
    def __init__(self):
        super(Acce, self).__init__()
        self.sensorEnabled = False

    def get_orientation(self, dt):
        val = gyroscope.orientation
```

self.ids.label1.text = "X: " + str(val[0])
self.ids.label2.text = "Y: " + str(val[1])
self.ids.label3.text = "Z: " + str(val[2])

def pressed1(self):
    try:
        if not self.sensorEnabled:
            gyroscope.enable()
            Clock.schedule_interval(self.get_orientation,
                                    1 / 20.)

            self.sensorEnabled = True
            self.ids.button1.text = "Stop"
        else:
            gyroscope.disable()
            Clock.unschedule(self.get_orientation)

            self.sensorEnabled = False
            self.ids.button1.text = "Start"
    except NotImplementedError:
        import traceback; traceback.print_exc()
        self.ids.status.text = "Gyroscope is not supported for your platform"

class e4App(App):
    def build(self):
        return Acce()

if __name__ == '__main__':
    e4App().run()

How it works...

The KV file that we are using is similar to the last recipe's KV file. We define four labels and the start button.

In the Python code, the first three lines are usual to import kivy and the widgets. The fourth line:

    from kivy.clock import Clock

We are importing the clock, which is used to get the accelerometer value in a time lap because, as we already said, this input is continuously providing data. The fifth line:

    from plyer import gyroscope
This is importing the package `plyer`, which is the one that makes it possible to retrieve the data from an iOS or Android device. The sixth line:

```python
class Accle(Widget):
```

It is the line where the class for the rule starts and its name must be the same as in the Kivy rule. The seventh, eighth, and ninth lines are:

```python
def __init__(self):
    super(Accle, self).__init__()
    self.sensorEnabled = False
```

Those are where we define the methods that initialize the class and put the accelerometer on standby to be used later. The tenth, eleventh, twelfth, thirteenth, and fourteenth lines are as follows:

```python
def get_orientation(self, dt):
    val = gyroscope.orientation
    self.ids.label1.text = "X: " + str(val[0])
    self.ids.label2.text = "Y: " + str(val[1])
    self.ids.label3.text = "Z: " + str(val[2])
```

Those lines are allowing us to retrieve data from the gyroscope and enabling us to modify the text of the first three labels with the data of the orientation. The fifteenth line:

```python
def pressed1(self):
```

It is where we define the method performed when the button is pressed. In the sixteenth line, we start a `try` sentence to start to get data from the gyroscope. With the seventeenth line, we can open an `if` sentence for the case that the gyroscope is unable, enable it. The eighteenth line:

```python
Clock.schedule_interval(self.get_orientation, 1 / 20.)
```

In this line, we can get the intervals of the orientation by calling the `get_orientation` method. The next class is the usual one to display our Kivy interface.

### See also

If you want to run the code in a mobile device, see Chapter 9, *Kivy for Mobile Devices*. 
The differences between the touch and motion events

There is a key difference between touch and motion events. A motion event is a continuous succession of many touch events. However, we also know that a touch event always has the `pos` profile, namely position information. The motion event, however, is not dispatched throughout the widget tree.

Getting ready

In this recipe, we will use the Kv language for the design of the widgets, so we assume that the reader is familiar with the Kv language or did the lecture of the first chapter. Also, this recipe will use the common button widget for reference.

How to do it...

Use this recipe and follow these steps:

1. First, in the KV file, define a button:
   ```kivy
   <MyW>:
       Button:
           id: button1
           pos: 0,0
           text: 'Hello'
   ```

2. In the class of the widget in the Python code, we need to override the method `on_touch_move`.

3. Change the button's position with the information in `touch.pos`:
   ```python
   import kivy
   kivy.require('1.9.0')

   from kivy.app import App
   from kivy.uix.widget import Widget

   class MyW(Widget):
       def on_touch_move(self, touch):
           if 'pos' in touch.profile:
               self.ids.button1.pos = touch.pos

   class e5App(App):
   ```
def build(self):
    return MyW()

if __name__ == '__main__':
e5App().run()

How it works...

Let's start with the KV file, the first line is the name of the rule, similar to the class in the Python file. The second line is the definition of the button widget, the third line is the ID of the button, which is necessary to do the call of the widget inside the Python code, the fourth line specifies the initial position of the button, and the fifth line is the definition of the text in the button.

In the Python code, the initial four lines are usual to use Kivy and the button widget. The fifth one:

class MyW(Widget):

It is where we define the class associated with our rule in the KV file. The sixth line:

def on_touch_move(self, touch):

Here, we start the declaration of the dispatched on_touch_move method; you must note the parameter touch in the declaration, as this parameter is necessary to call the event using the input. Now the seventh line:

    if 'pos' in touch.profile:

This is a verification line, because we need to be sure that the input device used in the platform, where we are running our code, supports the button profile. If this line is not present when the next one is performed, the app could crash. The eighth line:

    self.ids.button1.pos = touch.pos

This line is where we do the call to the profile position, which gives the information of where the touch occurs, and we changed the position of the button with the ID button1 with that information. The last lines of the Python code are usual to run and display our Kivy interface. Just remember that the initial part of the name of the class is:

class e5App(App):

It will be same as the KV file; the name of the KV file in this case is e5.kv.

The last thing to remark is that we do not limit the input event to the button, so the touch can occur anywhere in the window.
There's more...

Also, the motion events are compatible with the touch event. It is possible to add to our code to the class `MyW` a `on_touch_down` method like this:

```python
def on_touch_down(self, touch):
    if 'button' in touch.profile:
        self.ids.button1.text = touch.button
```

With this addition, now when you touch the text inside, the button is going to change to the information about which button is used.

See also

To get more details about widgets, see the recipes in Chapter 4, *Widgets*.

Recognizing touch shapes

There is a useful tool provided by Kivy that permits us to recognize the shape of the touch that is performed. In this recipe, we will go through the foundations of using it.

Getting ready

In this recipe, we will use the Kv language for the design of the widgets, so we assume that the reader is familiar with the Kv language. Also, this recipe will use the common button and label widgets for reference.

How to do it...

Follow these steps:

1. First, in the KV file, define a button and an empty label:
   ```
   <MyW>:
   
   Button:
   id: button1
   pos: 0,0
   text: 'Hello'

   Label:
   id: label1
   pos: 50, 200
   text: ''
   ```
2. In the class of the widget in the Python code, we need to override the method `on_touch_move`.

3. Change the button position to the information in `touch.pos`.

4. Change the text in the label when a rectangular shape is present:

   ```python
   import kivy
   kivy.require('1.9.0')

   from kivy.app import App
   from kivy.uix.widget import Widget
   from kivy.input.shape import ShapeRect

   class MyW(Widget):
       def on_touch_move(self, touch):
           if 'pos' in touch.profile:
               self.ids.button1.pos = touch.pos
           if isinstance(touch.shape, ShapeRect):
               self.ids.label1.text = 'My touch have a rectangle shape of size ' + str(touch.shape.width) + str(touch.shape.height)

   class e6App(App):
       def build(self):
           return MyW()
       if __name__ == '__main__':
           e6App().run()
   ```

**How it works...**

In the KV file, the first line as usual is the name of the rule. The second line is the definition of the button widget, the third line is the ID of the button, which is necessary to do the call of the widget inside the Python code, the fourth line specifies the initial position of the button, and the fifth line is the definition of the text in the button. The sixth line is the definition of the label widget, the seventh line is ID of the label, the eighth line is the initial position of the label, and the ninth line is the initial text of the label.

In relation to the Python code, the initial four lines are the usual lines to use Kivy and the button widget. Next, the fifth one:

   ```python
   from kivy.input.shape import ShapeRect
   ```
In this line, we import the information of ShapeRect to be compared with the touch performed in the app. The sixth line is:

```python
class MyW(Widget):
```

This is where we define the class associated with our rule in the KV file. The seventh line:

```python
def on_touch_move(self, touch):
```

Here, we start the declaration of the dispatched `on_touch_move` method; you must note the parameter touch in the declaration, and this parameter is necessary to call the event by the input. The eighth and ninth lines are:

```python
if 'pos' in touch.profile:
    self.ids.button1.pos = touch.pos
```

Those lines change the position of the button. The tenth line is:

```python
if isinstance(touch.shape, ShapeRect):
```

It is where the comparison between the imported shape and the shape of the touch is performed. The eleventh line will change the label text with the dimension of the touch if this is rectangular. The last lines of the Python code are the usual lines to run and display our Kivy interface. Just remember that the initial part of the name of the class is:

```python
class e6App(App):
```

It will be the same of the KV file; the name of the KV file in this case is `e6.kv`

### See also

If you want to run your interface, take a look at our recipe *Running your code*, and to get more details about widgets, see the recipes in Chapter 4, *Widgets*.

### Detecting multitapping

The multitapping detection is a useful tool in the development of an app. When more than a touch occurs in a zone, it is what we call a multitap.

### Getting ready

We will work with the `on_touch_down` method, so it could be useful to go through the recipe *Evolving to the touchscreen* in this chapter. Also, this recipe will use the common button and label widgets for reference.
How to do it...

Follow these steps in this recipe:

1. First, in the KV file, define a button and an empty label:
   ```
   <MyW>:
       Button:
           id: button1
           pos: 0,0
           text: 'Hello'

   Label:
       id: label1
       pos: 200, 200
       text: ''
   ```

2. In the class of the widget in the Python code, we need to override the method `on_touch_down`.

3. Change the text in the label when a double tap is present.

4. Change the text in the label when a triple tap is present:
   ```python
   import kivy
   kivy.require('1.9.0')
   
   from kivy.app import App
   from kivy.uix.widget import Widget
   
   class MyW(Widget):
       def on_touch_down(self, touch):
           if touch.is_double_tap:
               self.ids.label1.text = 'Touch is a double tap !
               - The interval is: ' + str(touch.double_tap_time)
           elif touch.is_triple_tap:
               self.ids.label1.text = "Touch is a triple tap !
               - The interval is: {0} \nand distance between
               previous is {1}".format(touch.triple_tap_time, touch.triple_tap_distance)
   
   class e7App(App):
       def build(self):
           return MyW()

   if __name__ == '__main__':
       e7App().run()
   ```
How it works...

We are again using our KV file with a simple button and empty label with the ID `button1` and `label1` respectively.

In the Python code, the initial four lines are usual to use Kivy and the button widget. The fifth line is:

```python
class MyW(Widget):
```

It is where we define the class associated with our rule in the KV file. The sixth line is:

```python
def on_touch_down(self, touch):
```

Here, we start the declaration of the dispatched `on_touch_down` method; you must note the parameter `touch` in the declaration and this parameter is necessary to call the event by the input. The seventh and eighth lines are:

```python
if touch.is_double_tap:
    self.ids.label1.text = 'Touch is a double tap ! - The interval is: ' + str(touch.double_tap_time)
```

These lines are where we detect if the tap is double. If it is true, we change the text of the label defined in the KV file with the string `Touch is a double tap! - The interval is: plus the time that the double tap takes`. The ninth and tenth lines:

```python
elif touch.is_triple_tap:
    self.ids.label1.text = "Touch is a triple tap ! - The interval is: {0} \nand distance between previous is {1}".format(touch.triple_tap_time, touch.triple_tap_distance)
```

Those lines change the text of the label if the touch in a triple tap. The last lines of the Python code are the usual lines to run and display our Kivy interface. Just remember that the initial part of the name of the class is:

```python
class e7App(App):
```

It will be similar to the KV file; the name of the KV file in this case is `e7.kv`.

Grabbing touch events

There are some moments or areas where we do not want the touch performed in the same way. Therefore, we can grab some inputs to give them special treatment.
Getting ready

We will continue working with the touch profile, so it could be useful to review the previous recipes for a deeper understanding, also we will use the same KV file with the common button and label widgets whereby you will find a detailed explanation of the KV file in the recipe Recognizing touch shapes.

How to do it...

This recipe following the next steps:

1. In the KV file, define a button and a empty label:

   ```
   <MyW>:
       Button:
           id: button1
           pos: 0,0
           text: 'Hello'

       Label:
           id: label1
           pos: 200, 200
           text: ''
   ```

2. In the class of the widget in the Python code, we need to override the method `on_touch_down`.

3. If the touch coincides with the button area, grab the touch.

4. In the class of the widget in the Python code, we also need to override the method `on_touch_up`.

5. If the touch is grabbed, change the text of the label and ungrab the touch.

6. Otherwise change the text of the label of that of a different text:

   ```python
   import kivy
   kivy.require('1.9.0')

   from kivy.app import App
   from kivy.uix.widget import Widget

   class MyW(Widget):
       def on_touch_down(self, touch):
           if self.ids.button1.collide_point(*touch.pos):
               touch.grab(self)
               return True

       def on_touch_up(self, touch):
   ```
if touch.grab_current is self:
    self.ids.label1.text = 'Click on the screen but over the Button'
    touch.ungrab(self)
    return True
else:
    self.ids.label1.text = 'Click on the screen'

class e8App(App):
    def build(self):
        return MyW()

if __name__ == '__main__':
e8App().run()

**How it works...**

Again, in the KV file, we have the definition of a button and an empty label.

In the Python code, the initial four lines are the usual lines to use Kivy and the button widget. The fifth line is:

    class MyW(Widget):

This is where we define the class associated with our rule in the KV file. The sixth line is:

    def on_touch_down(self, touch):

Here, we start the declaration of the dispatched `on_touch_down` method; you must note the parameter `touch` in the declaration and this parameter is necessary to call the event do it by the input. The seventh, eighth, and ninth lines are:

    if self.ids.button1.collide_point(*touch.pos):
        touch.grab(self)
        return True

In these lines, we check whether the touch falls in the area of the button, and if so, we grab the touch. In the tenth line, we define the dispatched `on_touch_up` to make the decision of what to do with the grabbed touch. In the eleventh line, there is the if statement where if the touch is grabbed, we change the label to the text 'Click on the screen but over the Button' in the thirteen line, we ungrab the touch:

    touch.ungrab(self)
In the fifteenth line, we have the `else` statement where we change the text of the label to 'Click on the screen'. This section is optional, but you're strongly encouraged to include it and your editor may want to discuss with you why you have not.

The last lines of the Python code are the usual lines to run and display our Kivy interface. Just remember that the initial part of the name of the class is:

```python
class e8App(App):
```

It will be the same as the KV file; the name of the KV file in this case is `e8.kv`.

## Recording gestures

A gesture is a particular succession of touches, which we will give some special significance to. Kivy eases the way to handle gestures.

### Getting ready

In this recipe, we will use some predefined graphics in the Kivy framework just to highlight the gesture, and you do not need any further knowledge about it. Also, we have a predefined gesture that is provided in this recipe to you.

### How to do it...

In this recipe, we have the next steps:

1. Call the gesture database.
2. Make a method to categorize the gesture.
3. Set a marker for the initial touch.
4. Record the touch.
5. Do the comparison with the predefined gesture.
6. Print the result:

```python
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.graphics import Color, Ellipse, Line
from kivy.gesture import Gesture, GestureDatabase

gdb = GestureDatabase()

class e8App(App):

    def __init__(self):
        super(e8App, self).__init__()

        # Predefined gesture
        self.gesture = gdb.str_to_gesture('eNq1l0tuI0cMhvd9EXsTofgmL6BsA/)

    def build(self):
        self.gesture = gdb.str_to_gesture('eNq1l0tuI0cMhvd9EXsTofgmL6BsA/)

        return Builder.load_file('e8.kv')

def on_gesture_begin(self, gesture):
    self.gesture += gesture

    # Initial touch marker
    with canvas:
        Color(1, 0, 0, 1)
        Ellipse(pos=(self.gesture[0].x, self.gesture[0].y), size=(100, 100))

        Line(points=gesture, width=5)

    return True
```

```python
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.graphics import Color, Ellipse, Line
from kivy.gesture import Gesture, GestureDatabase

gdb = GestureDatabase()

class e8App(App):

    def __init__(self):
        super(e8App, self).__init__()

        # Predefined gesture
        self.gesture = gdb.str_to_gesture('eNq1l0tuI0cMhvd9EXsTofgmL6BsA/)

    def build(self):
        self.gesture = gdb.str_to_gesture('eNq1l0tuI0cMhvd9EXsTofgmL6BsA/)

        return Builder.load_file('e8.kv')

def on_gesture_begin(self, gesture):
    self.gesture += gesture

    # Initial touch marker
    with canvas:
        Color(1, 0, 0, 1)
        Ellipse(pos=(self.gesture[0].x, self.gesture[0].y), size=(100, 100))

        Line(points=gesture, width=5)

    return True
```
def simplegesture(name, point_list):
    g = Gesture()
    g.add_stroke(point_list)
    g.normalize()
    g.name = name
    return g

class GestureBoard(Widget):

    def __init__(self, *args, **kwargs):
        super(GestureBoard, self).__init__()
        self.gdb = GestureDatabase()
        self.gdb.add_gesture(check)

    def on_touch_down(self, touch):
        userdata = touch.ud
        with self.canvas:
            Color(1, 1, 0)
            d = 30.0
            Ellipse(pos=(touch.x - d/2, touch.y - d/2),
                   size=(d, d)) userdata['line'] = Line(points=(touch.x, touch.y))
        return True

    def on_touch_move(self, touch):
        try:
touch.ud['line'].points += [touch.x, touch.y]
return True
except (KeyError) as e:
    pass

def on_touch_up(self, touch):
    g = simplegesture('', list(zip(touch.ud['line'].points[::2], touch.ud['line'].points[1::2])))
    print("gesture representation:", self.gdb.gesture_to_str(g))
    print("check":, g.get_score(check))
    g2 = self.gdb.find(g, minscore=0.70)
    print(g2)
    if g2:
        if g2[1] == check: print("check")
        self.canvas.clear()

class e9App(App):
    def build(self):
        return GestureBoard()

if __name__ == '__main__':
e9App().run()

How it works...

As usual, the first three lines are to use Kivy and the widgets. The fourth line is where we import predefined graphics to later represent the start of the gesture. In the fifth line, we import the gesture profile and the database structure. In the sixth line, we initialize the gesture database. The seventh line is read from the terminal using a code like this for a gesture with the figure of a check (check mark). Next, we define the method simplegesture:

def simplegesture(name, point_list):
    g = Gesture()
    g.add_stroke(point_list)
    g.normalize()
    g.name = name
    return g
This method returns a gesture from a list of points. In the class of the widget, we define the initial method where we start the gesture database and we add the check gesture. In the on_touch_down definition, we put a circle in the first touch and added this position to the list of touches. In the on_touch_move method we added the rest of the points of the gesture to the list of touches. In the on_touch_up method, which is triggered when the gesture is released, we convert the list of touches to a gesture using the method simplegesture. Also in this method, we have the line:

```python
    g2 = self.gdb.find(g, minscore=0.70)
```

In the preceding line, we evaluate if the gesture is the gesture database with a 70 percent tolerance and we print the result with the next lines in the terminal.

**There's more...**

Now, you have a way to record a gesture and you can feed the gesture database with different gestures. An important recommendation to do this is to separate the predefined gestures in to another file, which will contain only this information.

**See also**

To get more details about widgets, see the recipes in Chapter 4, Widgets, and for graphics, see Chapter 5, Graphics – Canvas and Instructions.
Where to buy this book

You can buy Kivy Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.