Kivy: Interactive Applications in Python

Create cross-platform UI/UX applications and games in Python

Roberto Ulloa

[PACKT] PUBLISHING

open source*
community experience distilled

Kivy: Interactive Applications in Python

Create cross-platform UI/UX applications and games in Python

Roberto Ulloa

[PACKT] open source*
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Kivy: Interactive Applications in Python

Credits

Author
Roberto Ulloa

Reviewers
Anai Arroyo B.
Andrés Vargas González
Javier de la Rosa
Hugo Solis

Acquisition Editor
James Jones

Commissioning Editor
Sruthi Kutty

Technical Editors
Ruchita Bhansali
Gauri Dasgupta
Monica John

Project Coordinator
Michelle Quadros

Proofreader
Amy Johnson

Indexer
Monica Ajmera Mehta

Graphics
Ronak Dhruv

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Roberto Ulloa has a diverse academic record in multiple disciplines within the field of Computer Science. He obtained an MSc from the University of Costa Rica and also taught programming and computer networking there. He then spent two years researching about cultural complexity at PhD level at the CulturePlex Lab of the University of Western Ontario.

He loves travelling and enjoys an itinerant life, living among different cultures and environments. He loves nature and has spent many months volunteering in Central and South America.

He currently lives in Kuala Lumpur, earning a living as a web developer in Python/Django and PHP/Wordpress. He constantly worries that the Internet has already become aware of itself and we are not able to communicate with it because of the improbability of it being able to speak Spanish or any of the 6,000 odd human languages that exist in this world.

Acknowledgments

I would like to thank Su, for not hesitating one second in encouraging and trusting my ability to write this book; for believing in me and motivating me with endless cups of coffee.

Javier de la Rosa, with whom I worked on my first Kivy project—the one that gave birth to the blog post that caught the attention of my publishers.

My technical reviewers, Anaí Arroyo, Javier de la Rosa, Hugo Solís and Andrés Vargas for their time and corrections.

My supervisor, Gabriela Barrantes, who has been a constant source of support and inspiration throughout my academic life.

My family and friends, for whom this book will be a surprise, and who've paid with the time that I didn't have to share with them.

The editorial personnel, for their patience in answering my questions.

Celina, for risking her Android to test my codes for the first time; for her constant motivation, support, and criticism even though I disagree that my Space Invaders look like bunnies and, if so, I still think they are terrifying space bunnies.

About the Reviewers

Anaí Arroyo is a PMI certified Project Manager who loves software development and is passionate about how technology can be used to improve the quality of people's life and volunteering as a way to contribute to make a positive difference.

Over the last years, she has worked in the Education field, collaborating in the design and development of Learning Management and Student Information Management systems.

Andrés Vargas González is currently pursuing a Master of Science in Computer Science through a Fulbright Fellowship at University of Central Florida (UCF). He received a Bachelor's degree in the same field from Escuela Superior Politécnica del Litoral (ESPOL) in Ecuador.

He is a member of the Interactive Systems and User Experience Lab at UCF. His current research is on machine learning techniques to reduce the time on gesture recognition in context. His previous works include enterprise multimedia distribution and exploring usability of multi-touch interfaces in Information Systems, which was tested on his DIY multi-touch surface. He is also interested in web applications development. He implemented some e-commerce solutions as well as Facebook applications in his home country and recently was working in the backend of an educational data resource system in Florida, USA.

Besides his academic and professional interests, he enjoys hiking high elevations, learning from different cultures, biking by the city, and finally, playing and watching soccer.

Javier de la Rosa is a full-stack Python developer since 2005, when he first met the Django web framework. During his years in Yaco, one of the main FLOSS-based companies in Spain, he leaded the Research and Development Team, participating in both European and national projects. Late in 2009, he started to collaborate with The CulturePlex Lab for Cultural Networks research, at the Western Unviersity in Canada, in his sparse time. As a result, he left Yaco in 2010 and joined the laboratory to lead and supervise technical and software developments. Today, he is still in charge of the developers team as well as conducting his own research on Big Culture, where he mixes his background as a BA and MA in Computer Sciences, Logics and Artificial Intelligence by the University of Seville, and his recent acquired skills as a 3rd year PhD student in Hispanic Studies at Western University in Canada. Currently, he just started his 1st year as a PhD in Computer Sciences, focusing on Graph Databases and Query Languages.

A regular collaborator of Open Source projects, he is the owner and main developer of qbe (`http://versae.github.io/qbe/`) and neo4j-rest-client (`https://github.com/versae/neo4j-rest-client`). In the academic field, he is author of several articles, as well as one of the writers of the book Programming Historian 2 (`http://programminghistorian.org/`). You can always contact him on Twitter (`@versae`) or GitHub under the nickname *versae*.

Hugo Solis is an assistant professor in the Physics Department at University of Costa Rica. His current research interests are computational cosmology, complexity and the influence of hydrogen on material properties. He has wide experience with languages including C/C++ and Python for scientific programming and visualization. He is a member of the Free Software Foundation and he has contributed code to some free software projects. Currently, he is in charge of the IFT, a Costa Rican scientific non-profit organization for the multidisciplinary practice of physics. (`http://iftucr.org`)

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Mobile devices have changed the way applications are perceived. They have increased in interaction types, the user expects gestures, multi-touches, animations, and magic-pens. Moreover, compatibility has become a must-have if you want to avoid the barriers imposed by major Operative Systems. Kivy is an Open Source Python solution that covers these market needs with an easy to learn and rapid development approach. Kivy is growing fast and gaining attention as an alternative to the established developing platforms.

This book introduces you into the Kivy world, covering a large variety of important topics related to interactive application development. The components presented in this book were not only selected according to their usefulness for developing state-of-the- art applications, but also for serving as an example of broader Kivy functionalities. Following this approach, the book covers a big part of the Kivy library.

Instead of giving a detailed description of all the functions and properties, it provides you with examples to understand their use and how to integrate the two big projects that come with the book. The first one, the comic creator, exemplifies how to build a user interface, how to draw vector shapes in the screen, how to bind user interactions with pieces codes and other components related to improve the user experience. The second project, Invaders Revenge, is a very interactive game that introduces you to the use of animations, scheduling of tasks, keyboard events, and multi-touch control.

Occasionally the book explains some technical but important Kivy concepts that are related to the order and strategies used to draw in the screen. These explanations give the readers some insights into the Kivy internals that will help them solve potential problems when they are developing their own projects. Even though they are not necessary for the comprehension of the main topics of this book, they will become important lessons when the readers are implementing their own applications.

The book keeps the readers attention by stating small problems and their solutions. The sections are short and straightforward, making the learning process constant. These short sections will also serve as a reference when they finish the book. However, serving as a reference doesn't prevent the text from achieving the main goal, which is teaching with bigger projects that connects the small topics. At the end of the book, the readers will feel comfortable to start their own project.

What this book covers

Chapter 1, *GUI Basics – Building an Interface*, introduces basic components and layouts of Kivy and how to integrate them through the Kivy Language.

Chapter 2, *Graphics – The Canvas*, explains the use of the canvas and how to draw vector figures on the screen.

Chapter 3, *Widget Events - Binding Actions*, teaches how to connect the interactions of the user through the interface with particular code inside the program.

Chapter 4, *Improving the User Experience*, introduces a collection of useful components to enrich the interaction of the user with the interface.

Chapter 5, *Invaders Revenge – An Interactive Multitouch Game*, introduces components and strategies to build highly interactive applications.

What you need for this book

This book requires a running installation of Kivy with all its requirements. The installation instructions can be found at `http://kivy.org/docs/gettingstarted/installation.html`.

Who this book is for

The book aims at Python developers who want to create exciting and interesting UI/UX applications. They should be already familiarized with Python and have a good understanding of some software engineering concepts, particularly inheritance, classes, and instances. That said, the code is kept as simple as possible and it avoids the use of very specific Python nuances. No previous experience of Kivy is required though some knowledge of event handling, scheduling, and user interface, in general, would boost your learning.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "`Missile` and `Shot` inherits from the same class called `Ammo`, which also inherits from `Image`. There is also the `Boom` class that will create the effect of explosion when any `Ammo` is triggered."

A block of code is set as follows:

```
# File name: hello.py
import kivy
kivy.require('1.7.0')

from kivy.app import App
from kivy.uix.button import Label

class HelloApp(App):
  def build(self):
    return Label(text='Hello World!')

if __name__=="__main__":
  HelloApp().run()
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Consider App as an empty window as shown in the following screenshot (**Hello World**! output)".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
GUI Basics – Building an Interface

Kivy emerges as a successor of PyMT (a library for multitouch applications) with a simple but ambitious goal in mind — same code for every commonplace platform: Linux / Windows / Mac OS X / Macosx / Android / iOS (*Mathieu Virbel*, `http://txzone.net/2011/01/kivy-next-pymt-on-android-step-1-done/`). This support is being extended to Raspberry Pi thanks to a founding campaign started by *Mathieu Virbel*, the creator of Kivy. Kivy was introduced in the EuroPython 2011, as a Python framework designed for creating natural user interfaces.

So, let's start creating user interfaces using one of its fun and powerful components, the **Kivy language** (`.kv`). The Kivy language helps us to separate the logic from the presentation. This is a fundamental engineering concept that helps to keep an easy and intuitive code. Nonetheless, it is possible to build a Kivy application using pure Python and Kivy as a library. We will also learn those concepts in later chapters because they allow us to modify interfaces dynamically.

This chapter covers all the basics for building a **graphical user interface** (**GUI**) in Kivy. Afterwards, you will be able to build practically any GUI you have in your mind, and even make them responsive to the size of window! The following is a list of all the skills that you're about to learn:

- Launching a Kivy application
- The Kivy language
- Creating and using widgets (GUI components)
- Basic properties and variables of the widgets
- Fixed, proportional, absolute, and relative coordinates
- Organizing GUIs through layouts
- Tips for achieving responsive GUIs

Apart from Python, this chapter requires some knowledge about Object-Oriented Programming (`http://en.wikipedia.org/wiki/Object-oriented_programming`) concepts. In particular, inheritance (`http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)`) and the difference between instances (`http://en.wikipedia.org/wiki/Instance_(computer_science)`) and classes (`http://en.wikipedia.org/wiki/Class_(computer_science)`) will be assumed. Before starting, you will need to install Kivy (The instructions can be found in `http://kivy.org/docs/installation/installation.html`). The book examples were tested on Kivy 1.7.0 but a more recent version should work as well.

At the end of this chapter, we will be able to build a GUI starting from a pencil and paper sketch. We will introduce the main project of the book — the Comic Creator, and implement the main structure of the GUI.

Hello World!

Let's put our hands on our first code. The following is yet another Hello World program:

```
1. # File name: hello.py
2. import kivy
3. kivy.require('1.7.0')
4.
5. from kivy.app import App
6. from kivy.uix.button import Label
7.
8. class HelloApp(App):
9.   def build(self):
10.     return Label(text='Hello World!')
11.
12. if __name__=="__main__":
13.   HelloApp().run()
```

This is merely a Python code. Launching a Kivy program is not different from launching any other Python application. In order to run the code, you just have to open a terminal (line of commands or console) and specify the command, `python hello.py --size=150x100` (`--size` is a parameter to specify the screen size). In the preceding code, the lines 2 and 3 verify if you have the appropriate version of Kivy installed in your computer.

> If you try to launch your application with an older Kivy version (say 1.6.0), an exception is raised for the specified version. There is no exception raised if you have a more recent version. Of course, backwards compatibility is desired but not always possible, and so you might face problems if you use a newer version.

We omit this statement in most of the examples inside the book, but you will be able to find it again in the online codes, which you can download, and its use is strongly encouraged in real life projects. The program uses two classes from the Kivy library (lines 5 and 6): `App` and `Label`. The `App` class is the starting point of any Kivy application. The following screenshot shows the window containing a Label with the **Hello World** text:
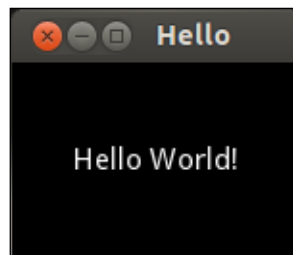
Hello World Output

The way we use the `App` class is through inheritance. `App` becomes the base class of `HelloApp` (line 8), the subclass or child class. In practice, this means that the `HelloApp` class has all the properties and methods of `App` in addition to whatever we define in the body (lines 9 and 10) of the `HelloApp` class.

In this case, the `HelloApp`'s body just modifies one of the existent `App`'s methods, the `build(self)` method. This method returns the window content. In this case, a simple `Label` saying `Hello World!` (line 10). Finally, the line 13 creates an instance of `HelloApp` and runs it.

So, is Kivy just another library for Python? Well, yes. But as part of the library, Kivy offers its own language to separate the logic from the presentation. For example you could write the preceding Python code in two separate files. The first file would then include the Python lines as shown in the following code:

```
14. # File name: hello2.py
15. from kivy.app import App
16. from kivy.uix.button import Label
17.
18. class Hello2App(App):
19.   def build(self):
20.     return Label()
21.
22. if __name__=="__main__":
23.   Hello2App().run()
```

The `hello2.py` code is very similar to `hello.py`. The difference is that the line 20 doesn't have the `Hello World!` message. Instead, the message has been moved to the `text` property in the second file (`hello2.kv`) which contains the Kivy language:

```
24. # File name: hello2.kv
25. #:kivy 1.7.0
26. <Label>:
27.   text: 'Hello World!'
```

How does Python or Kivy know that these files are related? This is quite important and tends to be confusing at the beginning. The key is in the name of the subclass of the `App`, that is, `HelloApp`.
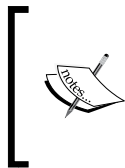
> The initial part of the `App`'s subclass name has to coincide with the name of the Kivy file. For example, if the definition of the class is `class FooApp(App)`, then the name of the file must be `foo.kv` and it must be in the same directory of the main file (the one that executes the `App`'s `run()` method).

Once that consideration is included, this example can be run in the same way we ran the `hello.py`. We just need to be sure that we are calling the new main file (`hello2.py`), `python hello2.py --size=150x100`.

This is your first contact with the Kivy language, so let's go slowly. The `#:Kivy 1.7.0` line of the `hello2.kv` code tells Python the minimal Kivy version that should be used. The line does the same that the lines 2 and 3 did in the `hello.py` code. The instructions that start with `#:` in the header of a Kivy language are called directives. We will also be omitting the version directive along the book, but remember to include it in your projects.

The `<Label>:` rule (line 26) indicates that we are going to modify the `Label` class by setting `'Hello World!'` in the `text` property (line 27). This code generates the same output that was shown in the previous screenshot. There is nothing you can't do using pure Python and importing the necessary classes from the Kivy library as we did in the first example (`hello.py`). However, the separation of the logic from the presentation results in simpler and cleaner code. Therefore, this book explains all the presentation programming through the Kivy language, unless dynamic components are added.

You might be worrying that modifying the `Label` class affects all the instances we create from `Label`, because, they will all contain the same `Hello World` text. That is true. Therefore, in the following section, we are going to learn how to directly modify specific instances instead of classes.

Basic widgets – labels and buttons

In the previous section, we were already using the `Label` class, which is one of the widgets that Kivy provides. Widgets are the little interface blocks that we use to set up the GUI. Kivy has a complete set of widgets including buttons, labels, checkboxes, dropdowns, and so on. You can find them all in the Kivy API `kivy.uix` (`http://kivy.org/docs/api-kivy.html`) under the package kivy.uix.

It's a good practice to create your own Widget for your applications instead of using the Kivy classes directly as we did in `hello2.kv` (line 26). The following code shows how to do that through inheritance:

```
28. # File name: widgets.py
29. from kivy.app import App
30. from kivy.uix.widget import Widget
31.
32. class MyWidget(Widget):
33.   pass
34.
35. class WidgetsApp(App):
36.   def build(self):
37.     return MyWidget()
38.
39. if __name__=="__main__":
40.   WidgetsApp().run()
```

In line 32 of the preceding code (`widgets.py`), we inherit from the base class `Widget` and create the subclass `MyWidget`, and, in line 37, we instantiated `MyWidget` instead of modifying the Kivy `Label` class directly as we did in `hello2.py`. The rest of the code is analogous to what we covered before. The following is the corresponding Kivy language code (`widgets.kv`):

```
41. # File name: widgets.kv
42. <MyWidget>:
43.  Button:
44.    text: 'Hello'
45.    pos: 0, 100
46.    size: 100, 50
47.    color: .8,.9,0,1
48.    font_size: 32
49.  Button:
50.    text: 'World!'
51.    pos: 100,0
52.    size: 100, 50
53.    color: .8,.9,0,1
54.    font_size: 32
```

Notice that now we are using buttons instead of labels. Most of the basic widgets in Kivy work in a very similar manner. In fact, `Button` is just a subclass of `Label` that includes more properties such as background color.

Compare the notation of line 26 (`<Label>:`) of `hello2.kv` with the line 43 (`Button:`) of the preceding code (`widgets.kv`). We used the class notation (`<Class>:`) for the `Label` (and for `MyWidget`) but another notation (`Instance:`) for `Button`. We just defined that `MyWidget` has two instances of `Button` (on line 43 and 49), and then we set the properties of those instances (the color is in RGBA format that stands for red, green, blue, and alpha/transparency).

The `size` and `pos` properties consist of fixed values, that is, the exact pixels on the window.
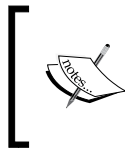
> Notice that the coordinate (0, 0) is at the bottom-left corner, that is, the Cartesian origin. Many other languages (including CSS) use the top-left corner as the (0, 0) coordinate, so be careful with this.

The following screenshot shows the output of the `widgets.py` and `widgets.kv` code files with some helpful annotations (on white color):

Creating our own widget

A couple of things could be improved in the previous code (`widgets.kv`). The first thing is that there are many repeated properties for the buttons such as `pos`, `color` and `font_size`. Instead of that, let's create our own `Button` as we did with `MyWidget`. The second is that the fixed position is quite annoying because the widgets don't adjust when the screen is resized because the position is fixed. Let's make the widgets more responsive:

```
55. # File name: widgets2.kv
56. <MyButton@Button>:
57.   color: .8,.9,0,1
58.   font_size: 32
59.   size: 100, 50
60.
61. <MyWidget>:
62.   MyButton:
63.     text: 'Hello'
64.     pos: root.x, root.top - self.height
65.   MyButton:
66.     text: 'World!'
67.     pos: root.right - self.width, root.y
```

In `widgets2.kv` we created (`<MyButton@Button>:`) and customized the `MyButton` class (as shown in lines 56 to 59) and `instances` (as shown in the lines 62 to 67).

Please note the difference between the way we defined `MyWidget` and `MyButton`. We need to specify `@Class` only if we didn't define the base class in the Python side as we did with `MyWidget` (line 32 of `widgets.py`). On the other hand, we had to define `MyWidget` in the Python side because we instantiated it directly (line 37 of `widgets.py`).

In this example, each `Button`'s position is responsive in the sense that they will always be displayed in the corners of the screen, no matter what the window size is. In order to achieve that, we need to use the `self` and `root` variables. You might be familiar with the variable `self`. As you have probably guessed, it is just a reference to the `Widget` itself. For example, `self.height` (line 64) has a value of 50 because that is the height of that particular `MyButton`. The `root` variable is a reference to the `Widget` class at the top of the hierarchy. For example, the `root.x` (line 64) has a value of 0 because that is the position in the X-axis of the `MyWidget` instance created on in line 37 of `widgets.py`. Since, the `MyWidget` instance is the only one in the `WidgetsApp`, it uses all the space by default; therefore, the origin is (0, 0). The x, y, width, and height are also the widgets properties.

Still, fixed coordinates are an inefficient way of organizing widgets and elements in the window. Let's move on to something smarter: layouts.

Layouts

No doubt that fixed coordinates are the most flexible way of organizing elements in an n-dimensional space; however, it is very time-consuming. Instead, Kivy provides a good set of layouts instead, which facilitate the work of organizing widgets. A `Layout` is a `Widget` subclass that implements different strategies to organize embedded widgets. For example, one strategy could be organizing widgets in a grid (`GridLayout`).

Let's start with a simple `FloatLayout` example. It works very similar to the way we organize widgets directly inside another `Widget`, except that now we can use proportional coordinates (proportions of the total size of the window) rather than fixed coordinates (exact pixels). This means that we don't need the calculations we did in the previous section with `self` and `root`. The following is the Python code:

```
68. # File name: floatlayout.py
69.
70. from kivy.app import App
71. from kivy.uix.floatlayout import FloatLayout
72.
73. class FloatLayoutApp(App):
74.   def build(self):
75.     return FloatLayout()
76.
77. if __name__=="__main__":
78.   FloatLayoutApp().run()
```

There is nothing new in the preceding code (`floatlayout.py`) except for the use of `FloatLayout` (on line 75). The interesting parts are in the Kivy language (`floatlayout.kv`):

```
79. # File name: floatlayout.py
80. <Button>:
```

```
81.    color: .8,.9,0,1
82.    font_size: 32
83.    size_hint: .4, .3
84.
85. <FloatLayout>:
86.   Button:
87.     text: 'Hello'
88.     pos_hint: {'x': 0, 'top': 1}
89.   Button:
90.     text: 'World!'
91.     pos_hint: {'right': 1, 'y': 0}
```

In the `floatlayout.kv` code file, we use two new properties, `size_hint` and `pos_hint`, which work with the proportional coordinates with values ranging from 0 to 1; (0, 0) is the bottom-left corner and (1, 1) the top-right corner. For example, the `size_hint` on line 83 sets the width to 40 percent of the current window width and the height to 30 percent of the current window height. Something similar happens to the `pos_hint` but the notation is different: a Python dictionary where the keys (for example, `'x'` or `'top'`) indicate which part of the widget is referenced. For instance, `'x'` is the left border. Notice that we use the `top` key instead of `y` in line 88 and `right` instead of `x` in line 91. The `top` and `right` properties respectively reference the top and right edges of the `Button`, so it makes the positioning simpler. That doesn't mean we could have used `x` and `y` for both the axes. For example, something like `pos_hint: {'x': .85, 'y': 0}` on line 91. The `right` and `top` keys avoids some calculations and makes the code clearer. The next screenshot illustrates the output of the previous code with the available keys for the `pos_hint` dictionary:
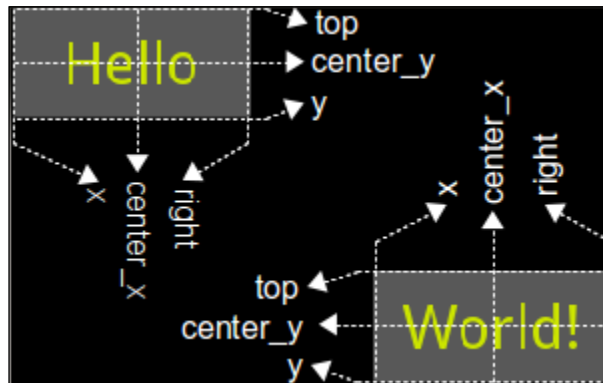
Using FloatLayout

The available `pos_hint` keys (`x`, `center_x`, `right`, `y`, `center_y`, and `top`) are useful for aligning to edges or centering. For example, `pos_hint: {'center_x':.5, 'center_y':.5}` would align a `Widget` in the middle, no matter what the size of the window is.

Could have we used `top` and `right` with the fixed positioning of `widgets2.kv` (in line 64 and 67)? Yes, we could; but notice that `pos` doesn't accept Python dictionaries (`{'x':0, 'y':0}`) that just lists of values (`0,0`). Therefore, instead of using the `pos` property, we have to use the `x`, `center_x`, `right`, `y`, `center_y`, and `top` properties directly. For example, instead of `pos: root.x, root.top - self.height`, we would have used the following code:

```
x: 0
top: root.height
```

Notice that these properties always specify fixed values (pixels) and not proportional ones.

> If we want to use proportional coordinates, we have to be inside a `Layout` (or an `App`), and use the `pos_hint` property.

If we are using a `Layout` instance, can we force the use of fixed values? Yes, but there can be conflicts if we are not careful with the properties we use. If we use any `Layout`, then `pos_hint` and `size_hint` will have the priority. If we want to use fixed positioning properties (`pos`, `x`, `center_x`, `right`, `y`, `center_y`, and `top`), we have to ensure that we are not using the `pos_hint` property. Secondly, if we want to use the `size`, `height`, or `width` properties, we need to give a `None` value to the `size_hint` axis we want to use with the absolute values. For example, `size_hint: (None, .10)` allows using the `height` property, but it keeps the width as 10 percent of the windows size. The following table summarizes everything we learned about the positioning and sizing properties. The first and second columns indicate the name of the property and its respective value. The third and fourth columns indicate if it is available for layouts and for widgets:

Property	Value	For layouts	For widgets
`size_hint`	A pair `[w, h]` where, `w` and `h` express a proportion (from 0 to 1 or `None`)	Yes	No
`size_hint_x` `size_hint_y`	A proportion from 0 to 1 or `None` indicating width (`size_hint_x`) or height (`size_hint_y`)	Yes	No
`pos_hint`	A dictionary with one x-axis key (`x`, `center_x`, or `right`) and one y-axis key (`y`, `center_y`, or `top`). The values are proportions from 0 to 1	Yes	No

size	A pair [w, h] where, w and h indicate fixed width and height in pixels	Yes, but set size_hint: (None, None)	Yes
width	A value indicating a fixed number of pixels	Yes, but set size_hint_x: None	Yes
height	A value indicating a fixed number of pixels	Yes, but set size_hint_y: None	Yes
pos	A pair [x, y] indicating a fixed coordinate (x, y) in pixels	Yes, but don't use pos_hint	Yes
x, right, or center_x	They have fixed number of pixels	Yes, but don't use x, right, or center_x in pos_hint	Yes
y, top, or center_y	The have fixed number of pixels	Yes, but don't use y, top, or center_y in pos_hint	Yes

We have to be careful because some of the properties behave different according to the layout we are using. Kivy currently has seven different layouts; six of them are briefly described in the following table. The left column shows the name of the Kivy Layout class and the right column describes briefly how they work:

Layout	Details
FloatLayout	This layout organizes the widgets with proportional coordinates with the size_hint and pos_hint properties. The values are numbers between 0 and 1 indicating a proportion to the window size.
Relative Layout	This layout operates in the same way as FloatLayout does, but the positioning properties (pos, x, center_x, right, y, center_y, and top) are relative to the Layout size and not the window size.
GridLayout	This layout organizes widgets in a grid. You have to specify at least one of the two properties: cols (for columns) or rows (for rows).
BoxLayout	This layout organizes widgets in one row or one column depending whether the value of property orientation is horizontal or vertical.
StackLayout	This layout is similar to BoxLayout but it goes to the next row or column when it runs out of space. In this layout, there is more flexibility to set the orientation. For example, 'rl-bt' organizes the widgets in right-to-left and bottom-to-top order. Any combination of lr (left to right), rl (right to left), tb (top to bottom), and bt (bottom to top) is allowed.
Anchor Layout	This layout organizes the widgets to a border or to the center. The anchor_x property indicates the x position (left, center or right), whereas anchor_y indicates the y position (top, center or bottom)

The seventh layout, which is the `ScatterLayout`, works similar to `RelativeLayout` but it allows multitouch gesturing for rotating, scaling, and translating. It is slightly different in its implementation so we will review it later. The Kivy API (`http://kivy.org/docs/api-kivy.html`) offers a detailed explanation and good examples on each of them. The behavioral difference of the properties depending on the `Layout` is sometimes unexpected but the following are some of the hints that will help you in the GUI building process:

- The `size_hint`, `size_hint_x`, and `size_hint_y` properties work on all the layouts but the behavior might be different. For example, `GridLayout` will try to take an average of the `x` hints and `y` hints on the same row or column respectively.

- It is advisable to use values from 0 to 1 with the `size_hint`, `size_hint_x`, and `size_hint_y` properties. However, you can also use values bigger than 1. Depending on the `Layout`, Kivy makes the `Button` bigger than the container or tries to recalculate a proportion based on the sum of the hints on the same axis.

- The `pos_hint` property works only in `FloatLayout`, `RelativeLayout`, and `BoxLayout`. In `BoxLayout`, only the x keys (`x`, `center_x`, and `right`) work in the vertical orientation and vice versa. An analogous rule applies for the fixed positioning properties (`pos`, `x`, `center_x`, `right`, `y`, `center_y`, and `top`).

- The `size_hint`, `size_hint_x`, and `size_hint_y` properties can always be set as `None` in favor of size, width, and height.

There are more properties and particularities of each `Layout`, but with the ones we have covered, you will be able to build almost any GUI you desire. In general, the recommendation is to use the layout as it is. Don't try to force the layout through an odd configuration of properties. Instead, it is better to use more layouts and embed them to reach our design goals. In the next section, we will teach you how to embed layouts, and we will offer a more comprehensive example of them.

Embedding layouts

The layouts studied in the previous section are subclasses of `Widget`. We have already been embedding widgets inside widgets since the beginning of this chapter and, of course, it won't matter if the widgets we are embedding are layouts as well. The following Python code is the base of a comprehensive example about embedding `Layouts`:

```
92. # File name: layouts.py
93. from kivy.app import App
94. from kivy.uix.gridlayout import GridLayout
95.
```

```
96. class MyGridLayout(GridLayout):
97.    pass
98.
99. class LayoutsApp(App):
100.   def build(self):
101.      return MyGridLayout()
102.
103. if __name__=="__main__":
104.    LayoutsApp().run()
```

There's nothing new in the preceding code, we just implemented the `MyGridLayout` class. The final result is shown first in the following screenshot with a few indications in different colors:
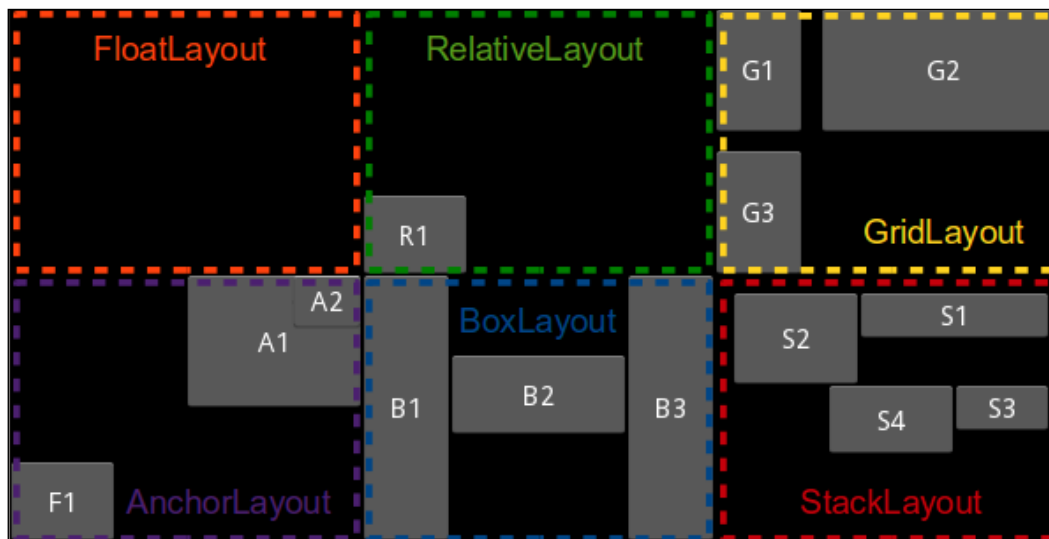
In the preceding screenshot all the types of Kivy layouts are embedded into a `GridLayout` of 2 rows by 3 columns. This is a big example, so we are going to study the corresponding Kivy language code (`layouts.kv`) in five fragments. Don't be overwhelmed by the amount of code, it is very straightforward. The following code is the fragment 1:

```
105. # File name: layouts.kv (Fragment 1)
106. <MyGridLayout>:
107.    rows: 2
108.    FloatLayout:
109.      Button:
110.        text: 'F1'
111.        size_hint: .3, .3
```

```
112.        pos: 0, 0
113.    RelativeLayout:
114.      Button:
115.        text: 'R1'
116.        size_hint: .3, .3
117.        pos: 0, 0
```

In the preceding code, `MyGridLayout` is first defined by the number of rows (line 107). Then we add the first two layouts, `FloatLayout` and `RelativeLayout` with one `Button` each. Both the buttons (**F1** and **R1**) have defined the property `pos: 0,0` (lines 112 and 117) but you can notice in the preceding screenshot (Embedding Layouts) that the `Button` **F1** (line 110) is in the bottom-left corner of the whole window, whereas the `Button` **R1** (line 115) is in the bottom-left corner of the `RelativeLayout`. The reason for this is that the coordinates in `FloatLayout` are not relative to the position of the layout. It is important to mention that this wouldn't have happened if we have used `pos_hint`, which always use the relative coordinates.

In the fragment 2, one `GridLayout` is added to `MyGridLayout` as shown in the following code:

```
118. # File name: layouts.kv (Fragment 2)
119.    GridLayout:
120.      cols: 2
121.      spacing: 10
122.      Button:
123.        text: 'G1'
124.        size_hint_x: None
125.        width: 50
126.      Button:
127.        text: 'G2'
128.      Button:
129.        text: 'G3'
130.        size_hint_x: None
131.        width: 50
```

In the preceding code, we define two columns (line 120) and a `spacing` of 10 (line 121) which separates the internal widgets by 10 pixels from each other. Also notice that in the previous screenshot (Embedding Layouts), the first column is thinner than the second column. We achieved this by setting the `size_hint_x` property to `None` and `width` to `50` of the buttons **G1** (line 122) and **G3** (line 128).

In the fragment 3, an `AnchorLayout` is added as shown in the following code:

```
132. # File name: layouts.kv (Fragment 3)
133.   AnchorLayout:
134.     anchor_x: 'right'
135.     anchor_y: 'top'
136.     Button:
137.       text: 'A1'
138.       size_hint: [.5, .5]
139.     Button:
140.       text: 'A2'
141.       size_hint: [.2, .2]
```

In the preceding code, we have specified `anchor_x` to `right` and `anchor_y` to `top` (line 134 and 135). You can notice in the previous screenshot (Embedding Layouts) how both the buttons (lines 136 and 139) of `AnchorLayout` have been placed in the top-right corner. This layout is very useful to embed other layouts inside it, for example top menu bars or side bars.

In the fragment 4, a `BoxLayout` is added as shown in the following code:

```
143. # File name: layouts.kv (Fragment 4)
144.   BoxLayout:
145.       orientation: 'horizontal'
146.       Button:
147.           text: 'B1'
148.       Button:
149.           text: 'B2'
150.           size_hint: [2, .3]
151.           pos_hint: {'y': .4}
152.       Button:
153.           text: 'B3'
```

The preceding code illustrates the use of `BoxLayout` in its horizontal `orientation`. Also, in lines 150 and 151 we use `size_hint` and `pos_hint` to move the button **B2** further up.

Finally, the fragment 5 adds a `StackLayout` as shown in the following code:

```
154. # File name: layouts.kv (Fragment 5)
155.   StackLayout:
156.     orientation: 'rl-tb'
157.     padding: 10
158.     Button:
159.       text: 'S1'
```

```
160.       size_hint: [.6, .2]
161.     Button:
162.       text: 'S2'
163.        size_hint: [.4, .4]
164.     Button:
165.       text: 'S3'
166.       size_hint: [.3, .2]
167.     Button:
168.       text: 'S4'
169.       size_hint: [.4, .3]
```

Here we have added four buttons of different sizes. To understand the rules applied to organize the widgets in the `rl-tb` (right-to-left and top-to-bottom) orientation (line 156) please refer back to the previous screenshot (Embedding Layouts). Also, you can notice that the `padding` property (line 157) adds `10` pixels of space between the widgets and the border of the `StackLayout`.

Our Project – comic creator

You now have all the necessary concepts to create any interface you want. This section describes the project that we will complete along this section itself and the following three chapters. The basic idea of the project is a comic creator, a simple application to draw a stickman. The following sketch is a wireframe of the GUI we have in mind.
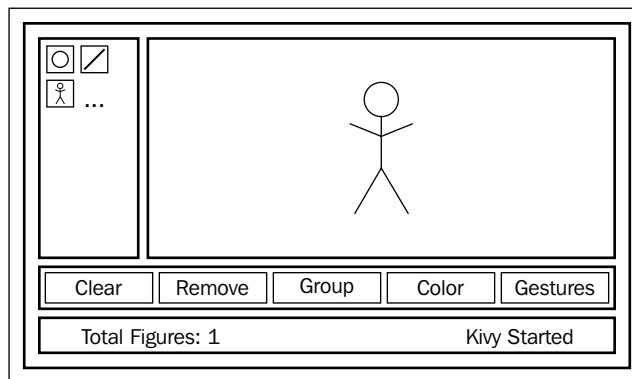
Comic creator sketch

We can distinguish several separate areas in the preceding sketch. Firstly, we need a drawing space (top-right) for our comics. We also need a tool box (top-left) with some drawing tools to draw our figures, and also some general options (second from bottom to top) such as clearing the screen, removing the last element, grouping elements, changing colors, and using the gestures mode. Finally, it will be useful to have a status bar (center-bottom) to provide some information to the user such as quantity of figures or the last action that has been performed. According to what we have learned along this chapter, there are multiple solutions to organize this screen but we will use the following:

- We'll use `AnchorLayout` for the toolbox area in the top-left corner
- We'll use `Gridlayout` of two columns for the drawing tools
- We'll use `AnchorLayout` for the drawing space in the top-right corner
- We'll use `RelativeLayout` to have a relative space to draw in
- We'll use `AnchorLayout` for the general options and status bar area at the bottom
- We'll use `BoxLayout` with vertical orientation to organize the general options on top of the status bar. We'll use also `BoxLayout` with horizontal orientation for the buttons of the general options, and again for the labels of the status bar.

We'll follow this structure by creating different files for each area: `comiccreator.py`, `comiccreator.kv`, `toolbox.kv`, `generaltools.kv`, `drawingspace.kv`, and `statusbar.kv`. Let's start with `comiccreator.py` as shown in the following code:

```
170. # File name: comiccreator.py
171. from kivy.app import App
172. from kivy.lang import Builder
173. from kivy.uix.anchorlayout import AnchorLayout
174.
175. Builder.load_file('toolbox.kv')
176. Builder.load_file('drawingspace.kv')
177. Builder.load_file('generaloptions.kv')
178. Builder.load_file('statusbar.kv')
179.
180. class ComicCreator(AnchorLayout):
181.    pass
182.
```

```
183. class ComicCreatorApp(App):
184.   def build(self):
185.     return ComicCreator()
186.
187. if __name__=="__main__":
188.   ComicCreatorApp().run()
```

We are explicitly loading some of the files with the `Builder.load_file` instruction (from line 175 to 178). There is no need to load the `comiccreator.kv` because it gets automatically loaded by the `ComicCreatorApp` name.

> The `Builder` class is in charge of loading and parsing all the Kivy language. The `load_file` method allows us to specify a file containing Kivy language rules that we want to include as part of the project.

For the `ComicCreator` we choose `AnchorLayout`. It is not the only option, but it illustrates more clearly within the code that the next level is composed of the `AnchorLayout` instances. You might wonder whether it might be possible to use a simple `Widget`. That would have been clear enough but unfortunately it is not possible because `Widget` doesn't honor the `size_hint` and `pos_hint` properties, which are necessary in the `AnchorLayout` internals.

The following is the code of the `comiccreator.kv`:

```
189. # File name: comiccreator.kv
190. <ComicCreator>:
191.   AnchorLayout:
192.     anchor_x: 'left'
193.     anchor_y: 'top'
194.     ToolBox:
195.       id: _tool_box
196.       size_hint: None,None
197.       width: 100
198.   AnchorLayout:
199.     anchor_x: 'right'
200.     anchor_y: 'top'
201.     DrawingSpace:
202.       size_hint: None,None
203.       width: root.width - _tool_box.width
204.       height: root.height - _general_options.height - _status_
bar.height
205.   AnchorLayout:
206.     anchor_x: 'center'
207.     anchor_y: 'bottom'
208.     BoxLayout:
209.       orientation: 'vertical'
210.       GeneralOptions:
211.         id: _general_options
```

```
212.          size_hint: 1,None
213.          height: 48
214.      StatusBar:
215.        id: _status_bar
216.         size_hint: 1,None
217.          height: 24
```

The preceding code follows the proposed structure for the comic creator. There are basically three `AnchorLayout` instances in the first level (lines 191, 198, and 205) and a `BoxLayout` that organizes the general options and the status bar (line 208).

The lines 197, 213 and 217 set the `width` of the `ToolBox` to `100` pixels, the `height` of the `GeneralOptions` to `48` pixels, and the height of the `StatusBar` to `24` pixels respectively. This brings up an interesting problem. We would like that the `DrawingSpace` uses all the remaining width and height of the screen (no matter what the windows size is). In other words, we want the drawing space as big as possible without covering the other areas (tool box, general options and status bar). In order to solve this, we introduced the use of `id` (in lines 195, 211, and 215) that allows us to refer to other components inside the Kivy language. On lines 203 and 204 we subtract the `tool_box` width to the `root` width (line 203) and the `general_options` and `status_bar` height to the `root` height (line 204). Accessing these attributes is only possible through the `id`'s we created, which can be used as variables inside the Kivy language.

Let's continue with the `toolbox.kv` as shown in the following code:

```
218. # File name: toolbox.kv
219. <ToolButton@ToggleButton>:
220.   size_hint: None,None
221.   size: 48,48
222.   group: 'tool'
223.
224. <ToolBox@GridLayout>:
225.   cols: 2
226.   padding: 2
227.   ToolButton:
228.     text: 'O'
229.     ToolButton:
230.     text: '/'
231.   ToolButton:
232.     text: '?'
```

We created a `ToolButton` class that defines the size of the drawing tools and also introduces a new Kivy `Widget:` `ToggleButton`. The difference from the normal `Button` is that it stays clicked until we click on it again. The following is an example of the toolbox with a `ToolButton` activated:
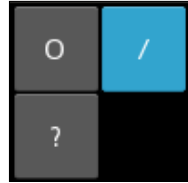
Toolbox area with an active ToggleButton

Moreover, it can be associated to other `ToggleButton` instances, so just one of them is clicked at a time. We can achieve this by assigning the same `group` property (line 222) to the `ToggleButton` instances we want to react together. In this case, we want all the instances of `ToolButton` instances to be part of the same group, so we set the group in the ToggleButton class definition (line 222).

On line 224, we implemented the `ToolBox` as a subclass of `GridLayout` and we added some character placeholders (`'O'`, `'/'`, and `'?'`) to the `ToolButton`. These placeholders will be substituted for something appropriate representations in the following chapters.

The following is the code of `generaloptions.kv`:

```
233. # File name: generaloptions.kv
234. <GeneralOptions@BoxLayout>:
235.    orientation: 'horizontal'
236.    padding: 2
237.    Button:
238.      text: 'Clear'
239.    Button:
240.      text: 'Remove'
241.    ToggleButton:
242.      text: 'Group'
243.    Button:
244.      text: 'Color'
245.    ToggleButton:
246.      text: 'Gestures'
```

In this case, when we used the `ToggleButton` instances (in lines 241 and 245), we didn't associate them to any `group`. Here, they are independent from each other and will just keep a mode or state. The preceding code only defines the `GeneralOptions` class, but there is no functionality associated yet. The following is the resulting screenshot of this area:
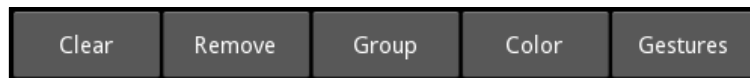
General Options area

The `statusbar.kv` file is very similar in the way it uses the `BoxLayout`:

```
247. # File name: statusbar.kv
248. <StatusBar@BoxLayout>:
249.   orientation: 'horizontal'
250.   Label:
251.     text: 'Total Figures: ?'
252.   Label:
253.     text: "Kivy started"
```

The difference is that it organizes labels and not buttons. The output is as shown in the following screenshot:

Status Bar area

The following is the code of `drawingspace.kv`:

```
254. # File name: drawingspace.kv
255. <DrawingSpace@RelativeLayout>:
256.   Label:
257.     markup: True
258.     text: '[size=32px][color=#3e6643]The[/color] [sub]Comic[/sub]
[i][b]Creator[/b][/i][/size]'
```

Apart from defining that `DrawingSpace` is a subclass of `RelativeLayout`, we introduce the Kivy markup, a nice feature for styling the text of the `Label` class. It works similar to the XML based languages. For example, in HTML (and XML based language) `<b>I am bold</b>` will specify bold text. First, you have to activate it (line 257) and then you just embed the text you want to style between `[tag]` and `[/tag]` (line 258). You can find the whole tag list and description in the Kivy API, in the documentation for `Label` at `http://kivy.org/docs/api-kivy.uix.label.html`.

In the previous example, `size` and `color` are self-explanatory; `sub` refers to the sub-indexed text; `b` refers to bold and `i` refers to italics. The following is the screenshot that shows the final GUI of our comic creator:
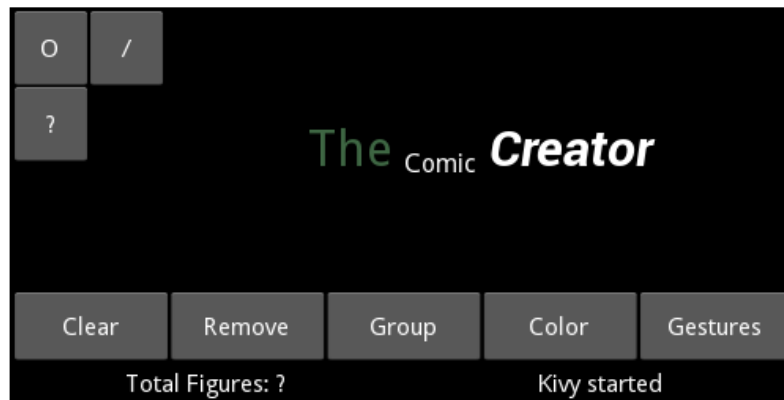
Final GUI of the Comic Creator

Along the following chapters we are going to add the respective functionality to this interface that, for now, consists of placeholders. However, it is exciting that with just a few lines of code, we implement a GUI that is ready to go for the rest of the book Comic Creator project. We will be working on its logic from now on.

Summary

This chapter covered all the basic, and some not so basic, concepts of Kivy. You learned how to configure classes, instances, and templates. The following is a list of Kivy elements we used in this chapter:

- We learned basic widgets such as `Widget`, `Button`, `ToggleButton`, and `Label`
- Layouts such as `FloatLayout`, `RelativeLayout`, `BoxLayout`, `GridLayout`, `StackLayout`, and `AnchorLayout`
- A number of properties such as `pos`, `x`, `y`, `center_x`, `center_y`, `top`, `right`, `size`, `height`, `width`, `pos_hint`, `size_hint`, `group`, `spacing`, `padding`, `color`, `text`, `font_size`, `cols`, `rows`, `orientation`, `anchor_x`, and `anchor_y`
- We also learned variables such as `self`, `root`, and id and markup tags such as `size`, `color`, `b`, `i`, and `sub`
- We used the `Builder` to load (the `load_file` method) extra Kivy language files.

There are much more elements that we can use inside the Kivy language. After this chapter we understand the general ideas and we should be able to use most of the elements available for the GUI design. There is, however, a very important and particular element that we haven't studied yet: the `canvas`, which mainly allow us to draw vector shapes (such as circles and lines for our project) in the screen. This will be the major topic of the next chapter.

2
Graphics – The Canvas

Any Kivy `Widget` contains a `Canvas` object. Be careful with the name because it might be confusing:

> A `Canvas` is not the place where we draw. Instead, a `Canvas` contains all the drawing instructions that will render the graphical representation of the `Widget`.

The coordinate space refers to the place where we draw, what you might have thought to be the canvas is in the first place. In this chapter, you are going to learn how to draw and manipulate the representation of the widgets through the instructions that we add to the `Canvas` instances:

- Draw basic geometric shapes (straight and curve lines, ellipses and polygons) through **vertex instructions**
- Using colors and rotating, translating, and scaling the coordinate space through the **context instructions**
- The difference between vertex and context instructions and how they complement each other
- The three different sets of instructions of the `Canvas` that we can use to modify the order of execution
- Storing and retrieving the current coordinate space context through `PushMatrix` and `PopMatrix`

Any `Widget` contains its own `Canvas` but all of them share the same coordinate space. This has consequences; for example, if we add a `Rotate` instruction to a specific `Canvas` (let's say the `Canvas` of a `Button`); it will also affect all the subsequent graphic instructions that are going to be displayed in the coordinate space. It doesn't matter if the graphics belong to canvases of different widgets. A context instruction changes the context of the coordinate space and we are going to learn strategies to control this.

Another important concept related to this is the `Widget`. A `Widget` is pretty much a place marker (with its position and size), but not necessarily a placeholder because the instructions of the `Canvas` of a `Widget` are not restricted to the specific area of the `Widget`, but to the whole coordinate space. At the same time, the coordinate space is not even restricted to the visual space of the Kivy window and this might also create some difficulties.

In this chapter, you will learn the available statements to draw and change the context of the coordinate space. More importantly, you will understand the canvas as a set of instructions, the implications of sharing a coordinate space, and the concept of widgets as place markers. In the last section of the chapter, we will illustrate the acquired knowledge within the comic creator. Here, you will also learn the most common technique to deal with the problem of sharing the same coordinate space. By the end, we will be on complete control of the graphics that are displayed on the screen.

Basic shapes

Before we start, let me introduce the Python code that we will use in most of this chapter's examples:

```
1. # File name: drawing.py
2. from kivy.app import App
3. from kivy.uix.relativelayout import RelativeLayout
4.
5. class DrawingSpace(RelativeLayout):
6.    pass
7.
8. class DrawingApp(App):
9.    def build(self):
10.      return DrawingSpace()
11.
12. if __name__=="__main__":
13.    DrawingApp().run()
```

We have created the subclass `DrawingSpace` from `RelativeLayout`. We could have chosen any `Widget` but we are going to integrate some of these ideas into our comic creator project later on, it is best to use `RelativeLayout` which the comic creator uses as drawing space.

There are two types of instructions that we can add to a `Canvas` instance. These instructions are represented by two base classes: `VertexInstructions` and `ContextInstructions`.

> The `VertexInstruction` subclasses allows us to draw vector shapes in the coordinate space. The `ContexInstruction` classes let us apply changes to the coordinate space context. For example, changing the colors or applying transformations (`Rotate`, `Translate`, and `Scale`). The coordinate space context describes the conditions in which the shapes (vertex instructions) are drawn in the coordinate space.

The following is the screenshot of the first example of this chapter (you should run that code with a screen size of 500x200: `python drawing.py --size=500x200`) that illustrates the use of vertex instructions:
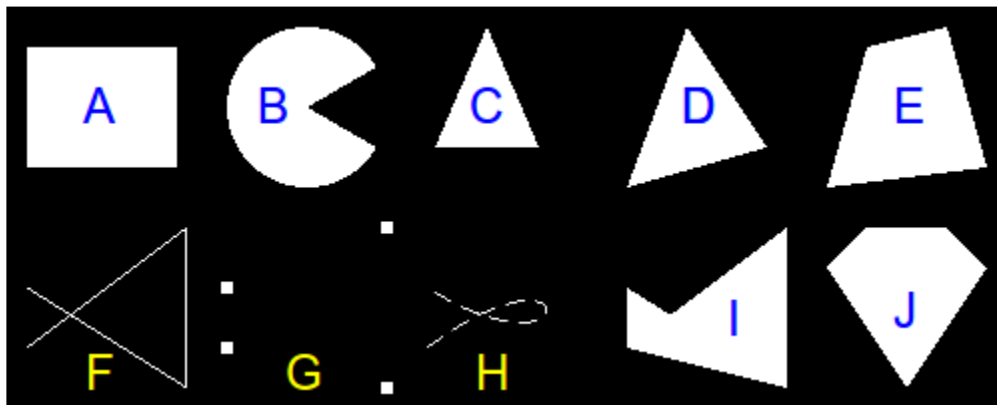
Vertex Instructions

The letters in blue and yellow in the preceding screenshot are references that will simplify the explanation (not part of the Kivy code). You can see 10 basic figures that we are learning to draw with `VertexInstruction` instances. Almost all the available `VertexInstruction` subclasses are represented in this example and we can create any 2D geometric shape with them. We will work on the with the Kivy language (`drawing.kv`) by small fragments, since we need to consider many details. Let's start with the shape A (rectangle) in the preceding screenshot:

```
14. # File name: drawing.kv (vertex instructions)
15. <DrawingSpace>:
16.    canvas:
17.      Rectangle:
18.        pos: self.x+10,self.top-80
19.        size: self.width*0.15, self.height*0.3
```

A `Rectangle` is a good starting point because it resembles the way we set properties in widgets. We just have to set the `pos` and `size` properties.

> All the values to specify the properties (for example, `pos` and `size`) of the vertex instructions are given in fixed values.

This means that we cannot use `size_hint` or `pos_hint` as we did in the *Chapter 1*, *GUI Basics: Building an Interface*. However, we can still use the properties of `self` (for example, lines 18 and 19) to achieve similar effects.

Let's proceed with the shape B (Pac-Man-like shape) as shown in the following code:

```
20.          Ellipse:
21.              angle_start: 120
22.              angle_end: 420
23.              pos: 110, 110
24.              size: 80,80
```

The `Ellipse` works very similar to the `Rectangle` but we have three new properties: `angle_start`, `angle_end`, and `segments`. The first two properties specify the initial and final angle of the ellipse. The angle 0° is north (or 12 o'clock) and they add up in the clock direction. That explains the angle 120° (90° + 30°) in line 21. It is not yet clear why the `angle_end` property is 420° (360° + (90°-30°)) and not just 60°. If you were to specify just 60°, Kivy would follow a counter clockwise direction painting the mouth of the Pac-Man instead of its body. Since, we need Kivy to follow the clockwise direction to paint the `Ellipse`, we have to be sure that our second angle is bigger than the first one.

Let's continue with the shape C (triangle) in the previous screenshot:

```
25.      Ellipse:
26.          segments: 3
27.          pos: 210,110
28.          size: 60,80
```

The triangle of shape C is actually another `Ellipse` that we can obtain; thanks to the `segments` property (line 26). Let's put it this way, if you have to draw an ellipse with three lines, the best you can get is a triangle. If you have four lines you can get a rectangle. You will need infinite lines for a complete `Ellipse`, but a computer cannot process that (and the screen doesn't have enough resolution to support this either). It becomes necessary to stop at some point. The default segments are 180 but you can modify the number of segment to obtain other shapes as shown above. Note that if you start with a circle, you will always get regular polygons (for example, a square if you specify just 4 `segments`).

We can analyze the shapes D, E, F, and G all together as shown in the following code:

```
29.      Triangle:
30.         points: 310,110,340,190,380,130
31.      Quad:
32.         points: 410,110,430,180,470,190,490,120
33.      Line:
34.         points: 10,30, 90,90, 90,10, 10,60
35.      Point:
36.         points: 110,30, 190,90, 190,10, 110,60
37.         pointsize: 3
```

The `Triangle`, `Quad`, and `Line` shapes work similarly. Their `points` property (lines 30, 32, and 34) indicates the corners of a triangle, a quadrilateral, or a line respectively. Line has many other properties but for now we just introduce its basic use. A `Point` is also similar to these three. It uses the `points` property (line 36) but in this case to indicate a set of points (look at the second row and first column of the screenshot). It uses the `pointsize` (line 37) property to indicate the size of the `Point`.

Now let's proceed with the shape as shown in the following code:

```
38.      Bezier:
39.         points: 210,30, 290,90, 290,10, 210,60
40.         segments: 360
41.         dash_length: 10
42.         dash_offset: 5
```

The `Bezier` is a curved line that uses the `points` property as a set of attractors of the curve line. There is a mathematical formalism behind the Bézier curves that we are not going to cover in this book because it is out of its scope. If you are interested, you can find enough information about it on Wikipedia at `http://en.wikipedia.org/wiki/Bézier_curve`). The points are attractors because the line does not touch all of them (it only touches the first and the last of them). Indeed, the points of the `Bezier` (line 39) are relatively equivalent to the ones of the `Line` (line 34) and the `Point` (line 36); they have just been translated 100px to the right. You can compare the different results of the trace in the previous screenshot (shapes F, G, and H). We have also included two other properties, `dash_length` (line 41) for the length of the dashes of the discontinuous line and `dash_offset` (line 42) for the distance between the dashes.

Let's cover the last shapes I and J as shown in the following code:

```
43.      Mesh:
44.        mode: 'triangle_fan'
45.        vertices: 310,30,0,0, 390,90,0,0, 390,10,0,0, 310,60,0,0
46.        indices: 0,1,2,3
47.      Mesh:
```

```
48.        mode: 'triangle_fan'
49.        vertices: 430,90,0,0, 470,90,0,0, 490,70,0,0, 450,10,0,0,
410,70,0,0
50.        indices: 0,1,2,3,4
```

We add two `Mesh` instructions (in lines 43 and 47). A Mesh is a compound of triangles and it has many applications in computer graphics and games. There is not enough space in this book to cover the advanced techniques for using this `VertexInstruction`, but we will learn its basics and be able to draw flat polygons. The property `mode` is set to `triangle_fan` (line 44), which means that the triangles of the mesh are filled with color instead of only drawn borders, for example.

The `vertices` property is a tuple of coordinates. For the purposes of this example we will just ignore all the 0s. That leaves us with four coordinates (or vertices) in line 45. Again, these points are relatively equivalent to the ones as shapes F, G, and H. Let's imagine how the triangles are created as we traverse them left to right on the vertex list using 3 vertex points each time. The shape I is composed by two triangles. The first triangle uses the first, second, and third vertices; and the second triangle uses the first, third, and fourth vertices. In general, if we are in the ith vertex of the list, a triangle is drawn using the first vertex, the (i-1)th vertex and the ith vertex. The final `Mesh` (shape J) presents another example. It contains three triangles that are surrounded by a blue line in the following screenshot:
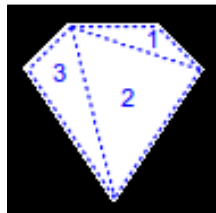
Triangles of the mesh

The `indexes` property contains a list with the same number of vertices (not counting the 0s) and instructs the order in which the vertices list is traversed, altering the triangles that compose the `Mesh`

So far, all the polygons that we have studied have been colored in. However, sometimes it is necessary to just draw the border of the polygon. In that case, we should use a `Line` instead. In principle, this seems easy for a basic shape such as a triangle, but how do we draw a circle with just points? The `Line` makes things easier for us here.

The next example will illustrate how you can build the set of figures in the following screenshot:

Line examples

The Python code for this example should be run in a screen size of 400 x 100: `python drawing.py --size=400x100`. The following is the `drawing.kv` code for the preceding screenshot:

```
51. # File name: drawing.kv (Line Examples)
52. <DrawingSpace>:
53.   canvas:
54.     Line:
55.       ellipse: 10, 20, 80, 60, 120, 420, 180
56.       width: 2
57.     Line:
58.       circle: 150, 50, 40, 0, 360, 180
59.     Line:
60.       rectangle: 210,10,80,80
61.     Line:
62.       points: 310,10,340,90,390,20
63.       close: True
```

In the preceding code, we added four `Line` instructions using very specific properties. The first `Line` (in line 54, shape A) is similar to our Pac-Man (line 20). The `ellipse` property (line 55) specify `x`, `y`, `width`, `height`, `angle_start`, `angle_end`, and `segments` respectively. The order of the parameters can be difficult to remember so always keep the Kivy API next to you (`http://kivy.org/docs/api-kivy.graphics.vertex_instructions.html`). We also set the `width` of the `Line` to make it thicker (line 56).

The second `Line` (in line 57, shape B) introduces a property that has no counterpart in the vertex instructions, that is, `circle`. The difference is that the first three parameters (line 58) define the center (`150`, `50`) and the radius (`40`) of the `circle`. The rest remains the same. The third `Line` (in line 59, shape C) is defined by a `rectangle` (line 60) and the parameters are simply `x`, `y`, `width`, `height`. The last `Line` (in line 61, shape D) ends being the most flexible way to define the polygons. We just specify the points (line 62), as many as we want. The `close` property (line 63) helps by connecting the first and last points.

[37]

We have covered most of the topics related to vertex instructions and now, you should be able to draw any geometrical shape in two dimensions with Kivy. If you want more details about each of the instructions, you should visit the Kivy API at `http://kivy.org/docs/api-kivy.graphics.vertex_instructions.html`. It is the turn of context instructions to decorate these boring black and white polygons.

Images, colors, and backgrounds

In this section, we will learn how to add images and colors to shapes and how to position graphics at a front or back level . We continue using the same Python code of the first section: `python drawing.py --size=400x100`. The following screenshot (Images and colors) shows the final result of this section:

Images and colors

The following is the corresponding `drawing.kv` code:

```
64. # File name: drawing.kv (Images and colors)
65. <DrawingSpace>:
66.   canvas:
67.     Ellipse:
68.       pos: 10,10
69.       size: 80,80
70.       source: 'kivy.png'
71.     Rectangle:
72.       pos: 110,10
73.       size: 80,80
74.       source: 'kivy.png'
75.     Color:
76.       rgba: 0,0,1,.75
77.     Line:
78.       points: 10,10,390,10
79.       width: 10
80.       cap: 'square'
81.     Color:
82.       rgba: 0,1,0,1
83.     Rectangle:
```

```
84.        pos: 210,10
85.        size: 80,80
86.        source: 'kivy.png'
87.    Rectangle:
88.        pos: 310,10
89.        size: 80,80
```

This code starts with an `Ellipse` (line 67) and a `Rectangle` (line 71). We use the `source` property, which inserts an image to decorate the polygon. The image `kivy.png` is 80 x 80 pixels with a white background (without any alpha/transparency channel). The result is shown in the first two columns of the previous screenshot (Images and colors).

In line 75, we use the context instruction `Color` to change the color (with the `rgba` property: red, green, blue, and alpha) of the coordinate space context. This means that the next `VertexInstructions` will be drawn with the color changed by `rgba`. A `ContextInstruction` changes the current coordinate space context. In the previous screenshot, the blue bar at the bottom (line 77) has a transparent blue (line 76) instead of the default white (`1,1,1,1`) as seen in the previous examples. We set the ends shape of the line to a `square` with the `cap` property (line 80).

We change the color again in line 81. After that, we draw two more rectangles, one with the `kivy.png` image and other without it. In the previous screenshot (Images and color) you can see that the white part of the image has become as green as the basic `Rectangle` on the left. Be very careful with this. The `Color` instruction acts as a light that is illuminating the `kivy.png` image. This is why you can still see the Kivy logo on the background instead of it being all covered by the color.

There is another important detail to notice in the previous screenshot. There is a blue line that crosses the first two polygons in front and then crosses behind the last two. This illustrates the fact that the instructions are executed in order and this might bring some unwanted results. In this example we have full control of the order but for more complicated scenarios Kivy provides an alternative.

> We can specify three `Canvas` instances (`canvas.before`, `canvas`, and `canvas.after`) for each `Widget`. They are useful to organize the order of execution to guarantee that the background component remains in the background, or to bring some of the elements to the foreground.

The following `drawing.kv` file shows an example of these three sets (lines 92, 98, and 104) of instructions:

```
90. # File name: drawing.kv (Before and After Canvas)
91. <DrawingSpace>:
92.   canvas.before:
93.     Color:
94.       rgba: 1,0,0,1
95.     Rectangle:
96.       pos: 0,0
97.       size: 100,100
98.   canvas:
99.     Color:
100.       rgba: 0,1,0,1
101.     Rectangle:
102.       pos: 100,0
103.       size: 100,100
104.   canvas.after:
105.     Color:
106.       rgba: 0,0,1,1
107.     Rectangle:
108.       pos: 200,0
109.       size: 100,100
110.   Button:
111.     text: 'A very very very long button'
112.     pos_hint: {'center_x': .5, 'center_y': .5}
113.     size_hint: .9,.1
```

In each set, a `Rectangle` of different color is drawn (lines 95, 101, and 107). The following diagram illustrates the execution order of the canvas. The number on the top-left margin of each code block indicates the order of execution:
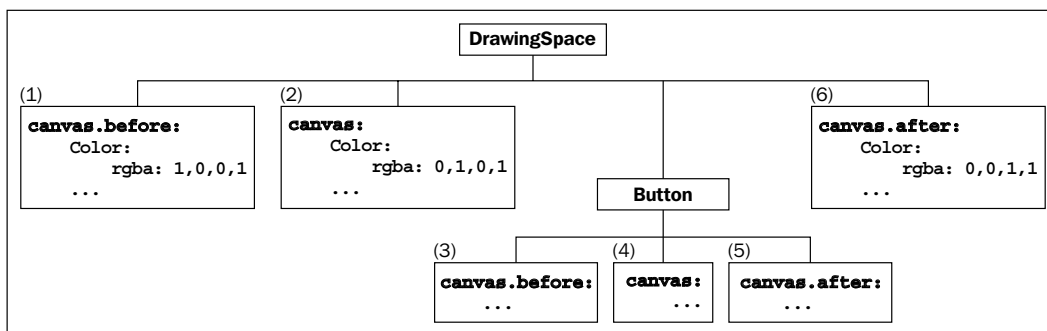
Execution order of the canvas

Please note that we didn't define any `canvas`, `canvas.before`, or `canvas.after` for the `Button`, but Kivy does. The `Button` is a `Widget` and it displays graphics on the screen. For example, the gray background is just a `Rectangle`. That means that it has instructions in its internal `Canvas` instances. The following screenshot shows the result (executed with `python drawing.py --size=300x100`):
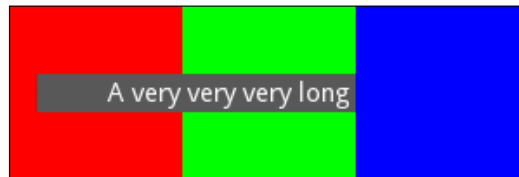
Before and after canvas

The graphics of the `Button` (the child) are covered up by the graphics of instructions in the `canvas.after`. But what is executed between `canvas.before` and `canvas`? It could be code of a base class when we are working with inheritance and we want to add instructions in the subclass that should be executed before the base class `Canvas` instances. A practical example of this will be covered when we apply them in the last section of this chapter in the comic creator project. The `canvas.before` will also be useful when we study how to dynamically add instruction to `Canvas` instances in *Chapter 4, Improving the User Experience*.

For now, it is sufficient to understand that there are three sets of instructions (`Canvas` instances) that provide some flexibility when we are displaying graphics on the screen. We will now explore some more context instructions related to three basic transformations.

Rotating, translating, and scaling

The `Rotate`, `Translate`, and `Scale` classes are the context instructions that are applied to the coordinate space, and indirectly to the vertex instructions. This may bring unexpected results if we forget that the coordinate space is of the size of the screen (actually bigger than that because there is no restriction in the coordinates and we can draw outside of the window), and this might affect all the subsequent components that are drawn. In this section, you will learn more about these problems; then, in the next section, we can analyze them more deeply and you will learn techniques that will facilitate working with the instructions.

Let's start with the new `drawing.kv` code as follows:

```
114. # File name: drawing.kv (Rotate, Translate, and Scale)
115. <DrawingSpace>:
116.     pos_hint: {'x':.5, 'y':.5}
117.     canvas:
```

```
118.      Rectangle:
119.        source: 'kivy.png'
120.      Rotate:
121.        angle: 90
122.        axis: 0,0,1
123.      Color:
124.        rgb: 1,0,0
125.      Rectangle:
126.        source: 'kivy.png'
127.      Translate:
128.        x: -100
129.      Color:
130.        rgb: 0,1,0
131.      Rectangle:
132.        source: 'kivy.png'
133.      Translate:
134.        y: -100
135.      Scale:
136.        xyz:(5,.5,0)
137.      Color:
138.        rgb: 0,0,1
139.      Rectangle:
140.        source: 'kivy.png'
```

In this code, the first thing we do is position the coordinate (0,0) of the `DrawingSpace` in the center of the screen (line 116). The displacement of the origin coordinate is only possible because `DrawingSpace` is a `RelativeLayout`. We create a `Rectangle` with the figure `kivi.png` that we modified to indicate the original X axis and Y axis. You can see the resulting image at the top-right corner of the following screenshot (executed with `python drawing.py --size=200x200`):
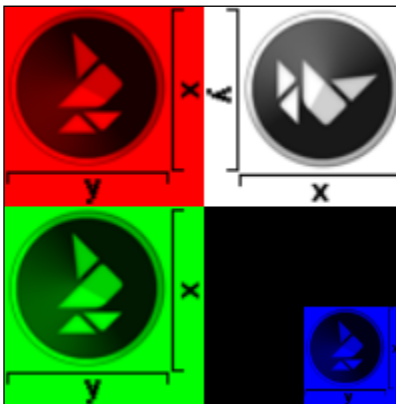
Rotate, translate, and scale

In line 120, we apply the `Rotate` instruction by 90° on the Z axis specified in line 122. The value is (`x`, `y` and, `z`), which means we could use any vector in the 3D space. Imagine nailing a pin in the bottom-left corner of the `DrawingSpace`. Then, we rotate it in the counter clock direction.
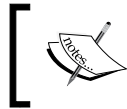
> By default, the pin nail of the rotation is always the coordinate (0, 0), but we can alter this behavior with the `origin` property.

The top-left section of the previous screenshot (Rotate, translate, and scale) shows the result after the rotation. We drew the same `Rectangle` with red `Color` (using the `rgb` property instead of the `rgba` property) to highlight it. After adding a rotation to the coordinate space context, we have also modified the relative X axis and Y axis. In line 128, this is taken into consideration, and the canvas is translated 100px to the bottom-left part using the X axis instead of the Y axis (as we may have expected). We drew the same `Rectangle` with green `Color`. You can notice that the image is still rotated and it will remain rotated as long as we don't bring the coordinate space context to its original angle.
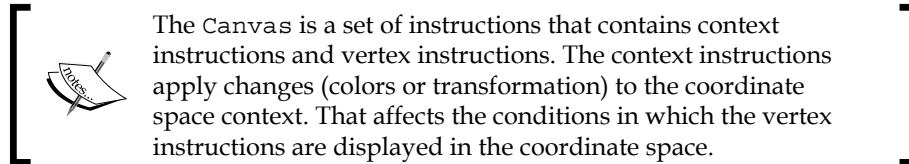
> The context instructions are persistent until we change them. This persistency is also broken when the context instructions were inside a `RelativeLayout` and we are now working outside of it.

In order to scale or zoom-out the image, we translate the coordinate space context (line 133) to use the bottom-right section of the previous screenshot. The scaling is done in line 135, where the image will be reduced to half the width and half the height. The `Scale` instruction reduces towards the (0, 0) coordinate. The question is where is the (0, 0) coordinate after the rotation and translations? First, we rotated the axis (line 120), so the X axis is vertical and the Y axis is horizontal. After translating down (line 128) and right (line 133), the (0, 0) coordinate is in the bottom right corner with the X axis being the vertical one and the Y axis the horizontal one. You can notice that `Scale` uses proportions to the current size of the coordinate space context and not the original size. For example, to recover the original size we should use `xyz: (2, 2, 0)` and not simply `xyz: (1, 1, 0)`.

In this section, we learned how to use basic transformations, but it can be summarized as follows:

> The `Canvas` is a set of instructions that contains context instructions and vertex instructions. The context instructions apply changes (colors or transformation) to the coordinate space context. That affects the conditions in which the vertex instructions are displayed in the coordinate space.

Using the acquired knowledge, we will now add a stickman to our project in the final section of the chapter. We will introduce two important instructions, `PushMatrix` and `PopMatrix`, to deal with the issues of widgets sharing the same coordinate space.

Comic creator – PushMatrix and PopMatrix

It is time to insert a few graphics to the project we started in the *Chapter 1*, *GUI Basics: Building an Interface*. Previously, we have insisted on the following two important things regarding the coordinate space:

- The coordinate space is not restricted to any position or size. It usually has its origin in the bottom-left corner of the screen. We avoided this in the last example by using `RelativeLayout`, which internally performs a translation to the `pos` property of the `Widget`.
- Once the coordinate space context is transformed by any instruction, it stays like that until we specify something different. `RelativeLayout` also addresses this problem with a two context instructions that we are going to study in this section: `PushMatrix` and `PopMatrix`.

We use `RelativeLayout` again in this section, but we will also explain alternatives to it whenever we deal with any other type of `Widget`. We will add a new file (`comicwidgets.kv`) to our project. In the `comicreator.py` we need to add our new file to the `Builder` as follows:

```
Builder.load_file('comicwidgets.kv')
```

The file `comicwidgets.kv` will contain special widgets that we will create for the project. In this chapter we are going to add the `StickMan` as follows:

```
141. # File name: comicwidgets.kv
142. <StickMan@RelativeLayout>:
143.    size_hint: None, None
144.    size: 48,48
```

```
145.    canvas:
146.      PushMatrix
147.      Line:
148.        circle: 24,38,5
149.      Line:
150.        points: 24,33,24,15
151.      Line:
152.        points: 14,5,24,15
153.      Line:
154.         points: 34,5,24,15
155.      Translate:
156.         y: 48-8
157.      Rotate:
158.        angle: 180
159.        axis: 1,0,0
160.      Line:
161.        points: 14,5,24,15
162.      Line:
163.        points: 34,5,24,15
164.      PopMatrix
```

In line 142, the StickMan subclass inherits from RelativeLayout to facilitate the
positioning and the use of the context instructions. The StickMan's size is defined as
48 x 48 pixels, but we are not limited to that since we now know how to scale. The
StickMan is composed of six lines that define head, body, left leg, right leg, left arm,
and right arm (from line 147 to 163).). Three different StickMan instances within one
image are shown in the following screenshot:
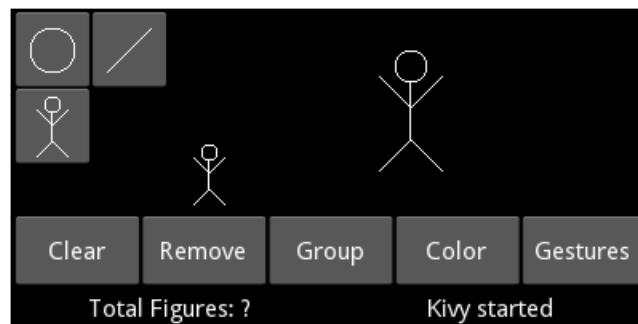
Comic creator

The first `StickMan` instance is part of the design of the last `ToolButton`; the other two appear in the drawing space. One of them is scaled. Notice that the code of the legs (from lines 151 to 154) is exactly same as the code for the arms (from lines 160 to 163). We translated it upwards (in lines 155 and 156) and rotated the canvas 180° on the X axis (from lines 157 to 159). The resulting code is probably more extensive than just calculating the points, but we saved ourselves some math in order to draw the `StickMan`.

Since we just translated and rotated the coordinate space context, we should undo these context changes so that everything will remain as it was at the beginning. Instead of adding more instruction to `Translate` and `Rotate` back the coordinate space context, we use two convenient Kivy instructions: `PushMatrix` and `PopMatrix`. At the beginning, we do a `PushMatrix` (line 146) that will save the current coordinate space context and at the end, we do the `PopMatrix` (line 164) to return the context to its original state.

> The `PushMatrix` instruction saves the current coordinate space context and the `PopMatrix` instruction retrieves the last saved coordinate space context. Therefore, the transformation instructions (`Scale`, `Rotate`, and `Translate`) surrounded by `PushMatrix` and `PopMatrix` won't affect the rest of the interface.
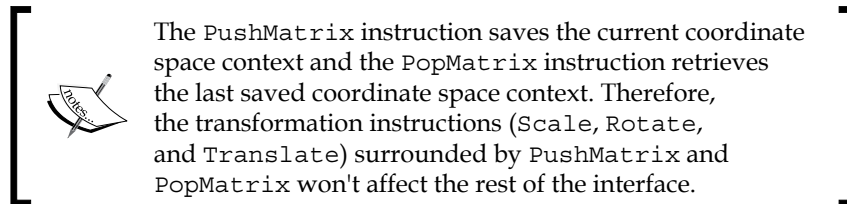
We will extend this approach to add shapes to the other two instances of `ToolButton` (circle and line) in the top-left part of the `ToolBox`. We add the following code to the `toolbox.kv`:

```
165. # File name: toolbox.kv
166. <ToolButton@ToggleButton>:
167.    size_hint: None,None
168.    size: 48,48
169.    group: 'tool'
170.    canvas:
171.       PushMatrix:
172.       Translate:
173.          xy: self.x,self.y
174.    canvas.after:
175.       PopMatrix:
176.
177. <ToolBox@GridLayout>:
178.    cols: 2
179.    padding: 2
180.    ToolButton:
181.       canvas:
```

```
182.        Line:
183.          circle: 24,24,14
184.    ToolButton:
185.      canvas:
186.        Line:
187.          points: 10,10,38,38
188.    ToolButton:
189.      StickMan:
190.        pos_hint: {'center_x':.5,'center_y':.5}
```

In the `Canvas` of the `ToolButton` class (line 166), we did a `PushMatrix` (line 171) in the `canvas` set of instruction. Then, the `Translate` instruction (line 172) moves the graphic instructions to the position of `ToolButton`, so we can use relative coordinates on each `ToolButton` (line 180 to 190). Finally, the `PopMatrix` (line 175) was added to the `canvas.after`. Let's review the execution order of the instructions sets (`Canvas` instances) using as an example the `ToolButton` that contains the circle (line 180). the following is its execution order:

1. The `canvas` of the `ToolButton` base class which has the `PushMatrix` and the `Translate` (line 170).

2. The `canvas` of the `ToolButton` subclass which has the circle (line 181).

3. The `canvas.after` of the base class which has the `PopMatrix` (line 174).

What we have just implemented with the `ToolButton` is the behind the curtains technique of the `RelativeLayout`.

> The `RelativeLayout` internally contains a `PushMatrix` and a `PopMatrix` instruction. This is why we can add instructions safely inside it without affecting the rest of the interface.

Let's scale our `StickMan` in the `DrawingSpace` and illustrate one more thing about the execution order of the canvas sets of instructions. The following is the code of `drawingspace.kv`:

```
191. # File name: drawingspace.kv
192. <DrawingSpace@RelativeLayout>:
193.    StickMan:
194.      pos_hint: {'center_x':.5,'center_y':.5}
195.      canvas.before:
196.        Translate:
197.          xy: -self.width/2, -self.height/2
198.        Scale:
199.          xyz: 2,2,0
200.    StickMan:
```

You might wonder why the second `StickMan` was neither scaled nor translated by the lines 196 and 198 in the previous screenshot (Comic creator)? The answer is not obvious. According to what we have studied before, the coordinate space context is global, so it should affect the second `Stickman`. The answer is not related to the `PushMatrix` and `PopMatrix` instructions inside the canvas of the `Stickman` class (lines 146 and 164) because both instructions are inside of the same set of instructions.

The solution is that the way we implemented the `ToolButton` follows the way `RelativeLayout` is implemented. `Stickman` inherits from `RelativeLayout`, so there is actually another `PushMatrix` in the `canvas.before` and its respective `PopMatrix` in the `canvas.after` of the `Stickman` class. The instructions in lines 196 to 199 are added after the `PopMatrix`. Therefore, the context is restored correctly on the `PushMatrix`. The instructions must be in the `canvas.before` because they are added after the existents instructions. In other words, if we were to add them in the canvas, the `StickMan` would be drawn before the translation and scaling.

In this section, we didn't change the `comiccreator.kv`, `generaloptions.kv`, and `statusbar.kv` files, so we are not presenting them again.

The context and vertex instructions are easy to understand. However, we must be very careful with the order of execution, and make sure to leave the coordinate space context in its original state after executing the desired vertex instructions. Finally, you should be aware that all you see on the screen is drawn by instructions within `Canvas` instances. For example, this includes `Label` texts and the `Button` backgrounds.

Summary

In this chapter we introduced many topics related to the use of the canvas. We covered the use of vertex and context instructions and how to manipulate the order of execution of instructions. You learned how to deal with transformation of the `Canvas`, either reversing all the transformations or using `RelativeLayout`. The following is a summary of components that you learned to use in this chapter:

- The `VertexInstructions` subclasses (and some of its properties): `Rectangle` (`pos` and `size`), `Ellipse` (`pos`, `size`, `angle_start`, `angle_end`, and `segments`), `Triangle` (`points`), `Quad` (`points`), `Point` (`points` and `pointsize`), `Line` (`points`, `ellipse`, `circle`, `rectangle`, `width`, `close`, `dash_lenght`, `dash_offset`, and `cap`), `Bezier` (`points`, `segments`, `dash_lenght`, and `dash_offset`), and `Mesh` (`mode`, `vertices`, and `indices`)
- The `source` property of `VertexInstructions` base class

- The three `Canvas` instances of the `Widget`: `canvas.before`, `canvas`, and `canvas.after`

- The context instructions (with some of their properties): `Color` (`rgba` and `rgb`), `Rotate` (`angle`, `axis`, and `origin`), `Translate` (`x`, `y`, and `xy`), `Scale` (`xyz`), `PushMatrix`, and `PopMatrix`

This list is comprehensive but there are still some remaining components. You can explore them in the Kivy API. The important part is that you learned the concepts behind the use of the `Canvas` class. The next chapter will focus on event handling and manipulating the Kivy objects directly from Python. This will allow us to connect the graphical user interface components with Python code and perform the desired actions on the comic creator.

3
Widget Events – Binding Actions

In this chapter you will learn how to integrate actions into the graphical user interface components; some of the actions will be associated with the canvas and others with the `Widget` management. We are going to control the user interaction with the interface, so we will have to handle actions dynamically. In this chapter, you will acquire the following skills:

- Connect different parts of the GUI through IDs and properties
- Override, bind, unbind, and create Kivy events
- Add widgets to other widgets dynamically
- Add vertex and context instructions to canvas dynamically
- Translate relative and absolute coordinates between the `Widget,` its parent and its window
- Use properties to keep the GUI updated with the changes

This is a very exciting chapter because you will start to see things happening on our screen and you will put together all the concepts acquired in the previous two chapters. All the basic functionality of our Comic Creator project will be ready by the end. This includes stickmen that can be dragged, circles and lines, sizeable circles and lines, clearing the widget space, removing the last added figure, grouping several widgets to drag them together, and keeping the status bar updated according to the last actions of the user.

Attributes, id and root

In *Chapter 1*, *GUI Basics – Building an Interface*, we distinguished between four main components for our Comic Creator: toolbox, drawing space, general options, and status bar. In this chapter these components will start to interact with each other and, therefore, we need to add some attributes to the classes of the project we created in the previous chapters. These attributes will reference different parts of the interface so they can interact. For example, the `Toolbox` class needs to have a reference to the `DrawingSpace` instance, so the `ToolButton` instances can draw their respective figures inside of it. The following diagram shows all the relationships that are created in the `comiccreator.kv` file:
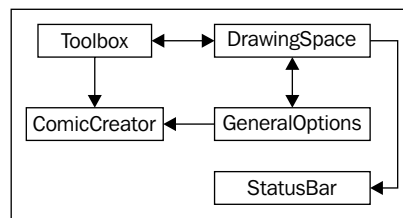
Internal References of the Comic Creator

We also learned in *Chapter 1*, *GUI Basics – Building an Interface* that **id** lets us reference other widgets in the Kivy language.

> However, ids by themselves are not intended to be used outside of the Kivy language: that is the Python code. We need to create some attributes in order to use the elements in the Python Code.

The following is the `comiccreator.kv` file of the ComicCreator project with some modifications:

```
1.  File Name: comiccreator.kv
2.  <ComicCreator>:
3.      AnchorLayout:
4.          anchor_x: 'left'
5.          anchor_y: 'top'
6.          ToolBox:
7.              id: _tool_box
8.              drawing_space: _drawing_space
9.              comic_creator: root
10.             size_hint: None,None
11.             width: 100
12.     AnchorLayout:
13.         anchor_x: 'right'
```

```
14.             anchor_y: 'top'
15.         DrawingSpace:
16.             id: _drawing_space
17.             status_bar: _status_bar
18.             general_options: _general_options
19.             tool_box: _tool_box
20.             size_hint: None,None
21.             width: root.width - _tool_box.width
22.             height: root.height - _general_options.height - _
status_bar.height
23.     AnchorLayout:
24.         anchor_x: 'center'
25.         anchor_y: 'bottom'
26.         BoxLayout:
27.             orientation: 'vertical'
28.             GeneralOptions:
29.                 id: _general_options
30.                 drawing_space: _drawing_space
31.                 comic_creator: root
32.                 size_hint: 1,None
33.                 height: 48
34.             StatusBar:
35.                 id: _status_bar
36.                 size_hint: 1,None
37.                 height: 24
```

IDs in lines 7, 16, 29, and 35 have been added. Following the previous diagram (Internal References of the ComicCreator), the ids are used to create the attributes in lines 8, 17, 18, 19, and 30.

> The names of the attribute and the id don't have to be different. In the previous code, we just added a '_' to the ids to distinguish them from the attributes. For instance, `_status_bar` is an id intended to be used in the Kivy language, and `status_bar` is an attribute intended to be used in Python. Both could have had the same name without causing a name conflict.

As an example, line 8 creates the attribute `drawing_space` which references the `DrawingSpace` instance. This means that the `Toolbox` instance can now access the `DrawingSpace` instance in order to draw figures on it. We can create attributes to other components of the Kivy language. One common component that we often want to have access is `root`. Lines 9 and 31 complete the referencing using the `root` to have access to it through the `comic_creator` attribute.

No more changes are needed in any other Comic Creator project file to create the attributes. Actually, at this point you can run the project as usual with `python comicreator.py` and obtain the same result of *Chapter 2*, *Graphics – The Canvas*. In the following sections, we are going to constantly use the created attributes to access or modify different parts of the interface.

Basic widget events – dragging the stickman

Basic `Widget` events correspond to touches on the screen. However, the concept of touch in Kivy is broader than might be intuitively assumed. It includes mouse events, finger touches, and magic pen touches. For the sake of simplicity, we will often assume in this chapter that we are using a mouse but it doesn't matter if we were using a touch screen (and the finger or magic pen instead). The following are the three basic `Widget` events:

- `on_touch_down`: When a new touch starts. For example, the action of pressing a button of the mouse or touching the screen.
- `on_touch_move`: When the touch is moved. For example, dragging the mouse or slide the finger over the screen.
- `on_touch_up`: When the touch ends. For example, releasing the mouse button or lift a finger from the screen.

Notice that an `on_touch_down` takes place each time before an `on_touch_move` and `on_touch_up` happens the bullet list also reflects the necessary execution order. Finally, `on_touch_move` could not happen at all if there was no dragging action. These events allow us to add drag capability to our `Stickman` so we can place it wherever we want. We have modified the header of `comicwidgets.kv` as follows:

```
38. # File name: comicwidgets.kv
39. #:import comicwidgets comicwidgets
40. <DraggableWidget>:
41.     size_hint: None, None
42.
43. <StickMan>:
44.     size: 48,48
45.     ...
```

The code now includes a rule for a new `Widget` called `DraggableWidget`. Line 41 `DraggableWidget` just deactivates the `size_hint`, so that we can use fixed sizes (line 44). The `size_hint: None, None` instruction has been removed from the `StickMan` because it will inherit from `DraggableWidget` in the Python Code. Line 39 is responsible for importing the respective `comicwidgets.py` file:

```
46. # File name: comicwidgets.kv
47. from kivy.uix.relativelayout import RelativeLayout
48. from kivy.graphics import Line
49.
50. class DraggableWidget(RelativeLayout):
51.     def __init__(self, **kwargs):
52.         self.selected = None
53.         super(DraggableWidget, self).__init__(**kwargs)
```

The `comicwidgets.py` file contains the new `DraggableWidget`. This class inherits from `RelativeLayout` (line 50). The attribute `selected` in line 52 will indicate if the `DraggableWidget` instance is selected or not (`__init__` is the constructor in Python and the right place to define class object attributes, this ofeten causes confusion to Java programmers). We have overloaded the 3 methods associated with the events we just studied (`on_touch_down`, `on_touch_move`, and `on_touch_up`). Each of these methods receives a `MotionEvent` as a parameter (`touch`) which contains useful information related with the event such as the coordinates of the touch.

Let's start with `on_touch_down`:

```
54.     def on_touch_down(self, touch):
55.         if self.collide_point(touch.x, touch.y):
56.             self.select()
57.             return True
58.         return super(DraggableWidget, self).on_touch_down(touch)
```

In line 55 we are using one of the most common methods in Kivy: `collide_point`. It allows us to detect if the event actually happens inside of a specific `DraggableWidget` by checking the coordinates of the touch.

This means that in principle every active `Widget` receives all the touch events (`MotionEvent`) that happen inside the window (coordinate space). It is up to the programmer to implement the logic that will discriminate between the possibility of a particular `Widget` doing something (in this case the `select` in line 56) with the event, or whether it will just let it pass (line 58).

The most common way of handling an event is using `collide_point`, but that doesn't mean that other criteria could be used. Kivy gives us absolute freedom in this. Line 55 provides the simplest case of checking whether the event occurred inside the `Widget`. If the coordinate of the event was actually inside the `Widget`, we call on the `select()` method which basically indicates that the figure has been selected (details explained at the end).

It is important to understand the returning value of an event (line 57) and also what calling the method of the base class means (line 58). The Kivy GUI has a hierarchical structure, so we always have a parent `Widget` (except if we are the root of the hierarchy).

The returning value tells the parent if we took care of the event or not by returning `True` or `False` respectively. We could also have some children widgets. If we want them to realize the event we must call on the method of the base class. If we don't call on that method, the children cannot realize that the event happened.

This means we need to be careful because we are in control of the widgets that receive the event. Finally, there is also the possibility to use the returning value of the `super` (base class reference) to find out whether one of the children has taken care of the event already.

The structure of lines 54 to 58 is probably the most common way of taking care of a basic event:

1. Make sure that the event happens inside the `Widget` (line 55).
2. Do what has to be done (line 56).
3. Return `True` indicating that the event was processed (line 57).
4. If the event falls outside of the `Widget`, then we propagate the event to the children and return the result (line 58).

Let's review the `select` method before we move on to the next event:

```
59.     def select(self):
60.         if not self.selected:
61.             self.ix = self.center_x
62.             self.iy = self.center_y
63.             with self.canvas:
64.                 self.selected = Line(rectangle=(0,0,self.
width,self.height), dash_offset=2)
```

First, we need to ensure that nothing has been selected before (line 60). If that is the case, we save the center coordinates of DraggableWidget (lines 61 and 62) and we dynamically draw a rectangle on its border (line 63 and 64) as illustrated in the following screenshot cut:

In *Chapter 2*, *Graphics – The Canvas*, we used the Kivy language to add shapes to the canvas. Now we are using Python code directly and we cannot use Kivy language syntax anymore. Notice that we keep the line instance in the selected attribute in line 64 because we will need to remove the rectangle. Also, the DraggableWidget instance will be self-aware if it is selected.

Let's continue with on_touch_move:

```
65.     def on_touch_move(self, touch):
66.         (x,y) = self.parent.to_parent(touch.x, touch.y)
67.         if self.selected and self.parent.collide_point(x - self.
width/2, y -self.height/2):
68.             self.translate(touch.x-self.ix,touch.y-self.iy)
69.             return True
70.         return super(DraggableWidget, self).on_touch_move(touch)
```

In this event, we control the dragging of the DraggableWidget. In line 67, we make sure that the DraggableWidget is selected. In the same line, we use collide_point again but this time we use the parent (DrawingSpace) instead of self. This is why the previous line (line 66) transformed the coordinates relative to the corresponding parent. We have to check the parent (DrawingSpace) because the Stickman can be dragged inside all the DrawingSpace and not just inside DraggableWidget itself. Another detail of line 66 is that we are checking the left corner of the future position of the DraggableWidget by subtracting half its width and height to the current touch (touch.x - self.width/2, touch.y - self.height/2).

If the conditions are True, we call on the translate method:

```
71.     def translate(self, x, y):
72.         self.center_x = self.ix = self.ix + x
73.         self.center_y = self.iy = self.iy + y
```

The method moves the `DraggableWidget` (x, y) pixels by assigning new values to `center_x` and `center_y`. It also updates the `ix` and `iy` that we created in the `select` method before. The `translate` method implementation might seem unnecessary but it will simplify translating several figures at a time with the **Group** Button of the general option area.

The last two lines (lines 69 and 70) of the `on_touch_move` method follow the same approach of that `on_touch_down` and the `on_touch_up` method (lines 77 and 78), presented as follows:

```
74.     def on_touch_up(self, touch):
75.         if self.selected:
76.             self.unselect()
77.             return True
78.         return super(DraggableWidget, self).on_touch_up(touch)
```

The `on_touch_up` event undoes the `on_touch_down` status. First, it checks if it is `selected` (instead of `collide_point`). If it is, then it calls the `unselected()` method:

```
79.     def unselect(self):
80.         if self.selected:
81.             self.canvas.remove(self.selected)
82.             self.selected = None
```

This method will dynamically remove the line instruction from the `canvas` (line 81) and set the variable `selected` back to `None` (line 82). Notice the different ways in which we add (line 63 and 64) and remove the line (line 81). Line 81 is a convenience based on the `with` Python statement, which allows us to simply add instructions to the canvas. It is equivalent to the call on `self.canvas.add(Rectantle(…))` with the advantage that it allows us to add many instructions at the same time. For example:

```
with self.canvas:
    Color(rgb=(1,0,0))
    Line(points=(0,0,5,5))
    Rotate()
    ...
```

There are two more lines of code in `comicwidgets.py`:

```
83. class StickMan(DraggableWidget):
84.     pass
```

Lines 83 and 84 just define our `StickMan` which now inherits from `DraggableWidget` (line 83) instead of from `RelativeLayout`.

In this section, you have learned the three basic Kivy events of any `Widget`. They are strongly dependent on the coordinates and so it is necessary to have a good control over them. For example, we couldn't avoid this in line 66 where we localized a coordinate relative to the parent. The next section explores the topic of localizing coordinates in much more detail.

Localizing coordinates – adding stickmen

In the last section, we used the `to_parent()` method (line 66) to translate the coordinates relative to the `DrawingSpace` to its parent.

> Please note, that we were inside `DraggableWidget` and the coordinates we received were relative to the parent (`DrawingSpace`).

These coordinates are convenient for `DraggableWidget` because we position it in the parent's coordinates. It allows us to use the coordinates in the parent's `collide_point`. This is no longer convenient when we want to check the coordinates on the parent's parent space or when we need to draw something directly on the canvas of a `Widget`.

Before studying more examples, let's summarize the theory.You learned that `RelativeLayout` is very useful because it is easier to think inside of a constraint space to localize our objects. The problems start when we need to translate the coordinates to other `Widget` area. Let us consider the following screenshot of a Kivy program:
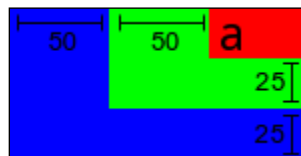
Three Embedded Relative Layouts

The code to generate this example is simple. It consists of three `RelativeLayouts` embedded into each other. The blue is parent of the green and the green is parent of the red. The `a` (on the top-right) is a `Label` inside the red `RelativeLayout` and it is located at the position (0, 0). The blue layout is of the size of the window (150x75 pixels).

The preceding screenshot includes some measures that help to explain the four methods of localizing coordinates that the `Widget` class provides:

- `to_parent()`: This method transforms the coordinates of the current `Widget` to the parent. For example, `a.parent.to_parent(a.x, a.y)` returns the coordinate of x relative to the green layout, which is `(50,25)`.

- `to_local()`: This method transforms the coordinates of the parent to the current widget. For example, `a.parent.to_local(50,25)` returns `(0,0)`, the coordinate of `a` relative to the red layout.

- `to_window()`: This method transforms the coordinates of the current `Widget` to the window. For example, `a.to_window(a.x, a.y)` returns the absolute coordinate of `a` which is `(100,50)`.

- `to_widget()`: This method transforms the absolute coordinates to the current widget. For example, `a.to_widget(100,50)` returns `(0,0)`, again the coordinate of `a` relative to the red layout.

The last two don't use the parent because Kivy assumes that the coordinates are always relative to the parent. There is also a Boolean parameter (called `relative`) which controls whether the coordinates are relative inside the `Widget`.

Let's study a real example in the Comic Creator project. We are going to add events to the tool box buttons, so that we can add figures to the drawing space. In this process, we will tackle a real case scenario in which we have to use one of the before-mentioned methods to localize our coordinates correctly to the `Widget`.

This code corresponds to the header of the `toolbox.py` file:

```
85. # File name: toolbox.py
86. import kivy
87. kivy.require('1.7.0')
88. import math
89. from kivy.uix.togglebutton import ToggleButton
90. from kivy.graphics import Line
91. from comicwidgets import StickMan, DraggableWidget
92.
93. class ToolButton(ToggleButton):
94.     def on_touch_down(self, touch):
95.         ds = self.parent.drawing_space
96.         if self.state == 'down' and ds.collide_point(touch.x,
touch.y):
97.             (x,y) = ds.to_widget(touch.x, touch.y)
98.             self.draw(ds, x, y)
99.             return True
```

```
100.            return super(ToolButton, self).on_touch_down(touch)
101.
102.    def draw(self, ds, x, y):
103.        pass
```

The structure in lines 94 to 100 is already familiar. Line 96 makes sure that
the `ToolButton` is in the `'down'` state and that the event happened in the
`DrawingSpace` instance (referenced by `ds`). Remember that the parent of the
`ToolButton` is `ToolBox`, and that we added an attribute that references the
`DrawingSpace` instance in `comiccreator.kv` at the beginning of the chapter.

The `draw` method is called in line 98 and it will draw the respective shapes according
to the derived classes (`ToolStickMan`, `ToolCircle`, and `ToolLine`). We need to
be sure that we send the right coordinates to the `draw` method. Therefore, before
calling on it we need to translate the absolute coordinates of the event to relative
coordinates with the `to_widget` event (line 97). We know that the coordinates
we received (`touch.x` and `touch.y`) are absolute because `ToolStickman` is not a
`RelativeLayout`, whereas the `DrawingSpace` (`ds`) is.

Let's continue studying the `toolbox.py` file and see how `ToolStickMan` actually
adds the `StickMan` instances:

```
104. class ToolStickman(ToolButton):
105.    def draw(self, ds, x, y):
106.        sm = StickMan(width=48, height=48)
107.        sm.center = (x,y)
108.        ds.add_widget(sm)
```

We create an instance of `Stickman` (line 106), we use the translated coordinates
(line 97) that were sent to the draw method in order to localize the center of the
`Stickman` and finally we add it to the `DrawingSpace` instance with the `add_widget`
method (line 108). At this point, we are able to add stickmen to the drawing space
and also drag them over it. We have the basics now. Let's learn how to bind and
unbind events dynamically.

Binding and unbinding events – sizing limbs and heads

In the previous two sections we override basic events to perform actions we wanted. In this section you will learn how to bind and unbind events dynamically. It was quite an easy job to add our `Stickman` because it is a `Widget`, but what about the graphics, the circle and the rectangle? We could create some widgets for them just as we did with the `Stickman` class but let's attempt something braver. Instead of just clicking on the drawing space, let's drag the mouse on its border to decide the size of the circle or line:

Using mouse to set the size

Once we finish the dragging (and we are satisfied with the size), let's dynamically create a `DraggableWidget` instance which contains the figures, so we can also drag them over the `DrawingSpace` instance. The following class diagram will help us to understand the whole inheritance structure of the `toolbox.py` file:
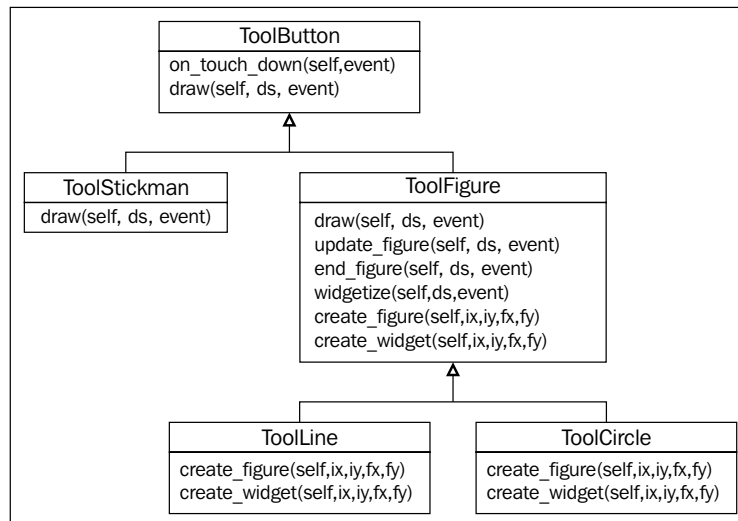
Class diagram of the tool buttons

The diagram includes the `ToolButton` and `ToolsStickman` which was explained in the last section but it also includes three new classes called `ToolFigure`, `ToolLine`, and `ToolCircle`.

The `ToolFigure` class has six methods. Let's start with a quick overview of these methods and then highlight the important and new parts:

1. `draw`: This method overrides the `draw` of `ToolButton` (lines 102 and Error! Reference source not found). The position where we touch down indicates the starting point of our figure, either the center for a circle or one of the ends for a line.

```
109. class ToolFigure(ToolButton):
110.     def draw(self, ds, x, y):
111.         (self.ix, self.iy) = (x,y)
112.         with ds.canvas:
113.             self.figure=self.create_figure(x,y,x+1,y+1)
114.         ds.bind(on_touch_move=self.update_figure)
115.         ds.bind(on_touch_up=self.end_figure)
```

2. `update_figure`: This method updates the end point of the figure when we are dragging. Either the end for a line or the radius (distance from the starting point to the end point) for the circle.

```
116.     def update_figure(self, ds, touch):
117.         if ds.collide_point(touch.x, touch.y):
118.             (x,y) = ds.to_widget(touch.x, touch.y)
119.             ds.canvas.remove(self.figure)
120.             with ds.canvas:
121.                 self.figure = self.create_figure(self.ix,
self.iy,x,y)
```

3. `end_figure`: This method indicates the final end point of the figure with the same logic as in `update_figure`. Also, we can put the final figure inside a `DraggableWidget` (see `widgetize`).

```
122.     def end_figure(self, ds, touch):
123.         ds.unbind(on_touch_move=self.update_figure)
124.         ds.unbind(on_touch_up=self.end_figure)
125.         ds.canvas.remove(self.figure)
126.         (fx,fy) = ds.to_widget(touch.x, touch.y)
127.         self.widgetize(ds,self.ix,self.iy,fx,fy)
```

4. `widgetize`: This method creates `DraggableWidget` and places the figure into it. It uses four coordinates that have to be localized correctly with the localization methods.

```
128.     def widgetize(self,ds,ix,iy,fx,fy):
129.         widget = self.create_widget(ix,iy,fx,fy)
130.         (ix,iy) = widget.to_local(ix,iy,relative=True)
131.         (fx,fy) = widget.to_local(fx,fy,relative=True)
132.         widget.canvas.add( self.create_figure(ix,iy,fx,fy))
133.         ds.add_widget(widget)
```

5. `create_figure`: This method is overridden by `ToolLine` (lines 139 to 140) and `ToolCircle` (lines 148 to 149). It creates the respective figure given four coordinates.

```
134.      def create_figure(self,ix,iy,fx,fy):
135.          pass
```

6. `create_widget`: This method is also overridden by `ToolLine` (lines 142 to 145) and `ToolCircle` (lines 151 to 155). It creates a respective positioned and sized `DraggableWidget` given four coordinates.

```
136.      def create_widget(self,ix,iy,fx,fy):
137.          pass
```

Most of the statements of the preceding methods have already been covered. The new topic on this code is the dynamically bind/unbind of events. The main problem we needed to solve is that we didn't want the `on_touch_move` and `on_touch_up` events active all the time. We needed to activate (bind) them from the moment the user start drawing (`on_touch_down` of `ToolButton` that calls on the method `draw`) until the user decides the size and do a touch up. Therefore, we have bound `update_figure` and `end_figure` respectively to the `on_touch_move` and `on_touch_up` event of the `DrawingSpace` when the method `draw` is called (lines 114 and 115). Also, we have unbound them when the user ends the figure on method `end_figure` (lines 123 and 124). Notice that we can unbind the same method that is being executed (`end_figure`) from the `on_touch_up` event. What we want to avoid is that the method is called on again.

There are a few other interesting things on this code that deserve some attention. In line 111 we have created two class attributes (`self.ix` and `self.iy`) to keep the coordinates of the initial touch. We use those coordinates each time we update the figure (line 121) and when we put the figure into a `Widget` (line 127).

We also use some of the localizing methods that we learned in the previous section. In lines 118 and 126 we have used `to_widget` to translate the coordinates to the `DrawingSpace` instance. The lines 130 and 131 use `to_local` for translating the coordinates to the `DraggableWidget`.
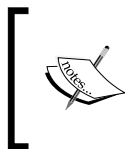
> `DraggableWidget` is instructed to translate the coordinates to its inner relative space with the parameter `relative=True` because `DraggableWidget` is relative and we are trying to draw inside it and not inside the parent: the drawing space.

There is some basic math involved in the calculation of the position and sizes of the figures and widgets. We have intentionally moved it to the deepest classes of the inheritance: `ToolLine` and `ToolCircle`. Following is their code, the last part of `toolbox.py`.

```
138. class ToolLine(ToolFigure):
139.     def create_figure(self,ix,iy,fx,fy):
140.         return Line(points=[ix, iy, fx, fy])
141.
142.     def create_widget(self,ix,iy,fx,fy):
143.         pos = (min(ix, fx), min(iy, fy))
144.         size = (abs(fx-ix), abs(fy-iy))
145.         return DraggableWidget(pos = pos, size = size)
146.
147. class ToolCircle(ToolFigure):
148.     def create_figure(self,ix,iy,fx,fy):
149.         return Line(circle=[ix,iy,math.hypot(ix-fx,iy-fy)])
150.
151.     def create_widget(self,ix,iy,fx,fy):
152.         r = math.hypot(ix-fx, iy-fy)
153.         pos = (ix-r, iy-r)
154.         size = (2*r, 2*r)
155.         return DraggableWidget(pos = pos, size = size)
```

The math involves concepts of geometry that escapes the scope of the book.

Finally, we are applying some changes to the `toolbox.kv`:

```
156. # File name: toolbox.kv
157. #:kivy 1.7.0
158. #:import toolbox toolbox
159. <ToolButton>:
160.     size_hint: None,None
161.     size: 48,48
162.     group: 'tool'
163.     canvas:
164.         PushMatrix:
165.         Translate:
166.             xy: self.x,self.y
167.     canvas.after:
168.         PopMatrix:
169.
170. <ToolBox@GridLayout>:
171.     cols: 2
172.     padding: 2
```

```
173.    tool_circle: _tool_circle
174.    tool_line: _tool_line
175.    tool_stickman: _tool_stickman
176.    ToolCircle:
177.        id: _tool_circle
178.        canvas:
179.            Line:
180.                circle: 24,24,14
181.    ToolLine:
182.        id: _tool_line
183.        canvas:
184.            Line:
185.                points: 10,10,38,38
186.    ToolStickman:
187.        id: _tool_stickman
188.        StickMan:
189.            pos_hint: {'center_x':.5,'center_y':.5}
```

The new classes `ToolCircle` (line 176), `ToolLine` (line 181), and `ToolStickMan` (line 186) have replaced the previous `ToolButton` instances. We have also created some attributes (lines 173, 174, and 175) that will be useful in *Chapter 4*, *Improving the User Experience*, when we use gestures to create figures.

Before we close up this section, let's take a deeper look to the `import` syntax in line 158. The word toolbox is repeated twice to tell Kivy to import all the classes of the `toolbox.py` file.
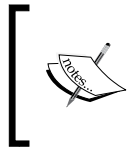
> We could have imported a specific class with the syntaxes `#:import Name package.ClassName` which is equivalent to the Python version: `from package.ClassName import ClassName as Name`.

Please note that it is the Kivy language file that is importing the Python file. This would be unnecessary if the Python file (`toolbox.py`) is imported first somewhere else. For example, we could also have imported the file in the `comicreator.py` file (where the `Builder` is calling this file).

Binding events in the Kivy language

In this section you will learn to bind events in the Kivy language. Potentially, we could have done this since the very beginning of the chapter when we started working with `DraggableWidget` but there is a difference. The difference is that if we use the Kivy language we can easily add the event to a specific instance and not to all the instances of the same class. For example, when we add the touch basic events to the `DraggableWidget`, all the instances created from it receive the implementation. In this sense, it resembles to externally binding an instance to its callback with the `bind` method. All that said, we can still make this binding of the class definition, if we added the code directly to the rule (`<DraggableWidget>:`) that defines the class. In any case, we are going to concentrate on new events specific for `Button` and `ToggleButton`.

The following is the code for `generaloption.kv`:

```
190. # File name: generaloptions.kv
191. #:import generaloptions generaloptions
192. <GeneralOptions>:
193.     orientation: 'horizontal'
194.     padding: 2
195.     Button:
196.         text: 'Clear'
197.         on_press: root.clear(*args)
198.     Button:
199.         text: 'Remove'
200.         on_release: root.remove(*args)
201.     ToggleButton:
202.         text: 'Group'
203.         on_state: root.group(*args)
204.     Button:
205.         text: 'Color'
206.         on_press: root.color(*args)
207.     ToggleButton:
208.         text: 'Gestures'
209.         on_state: root.gestures(*args)
```

Button has two extra events: on_press and on_release. The former is similar to on_touch_down and the latter to on_touch_up. However, in this case. we don't need to worry about calling on the collide_point method. We use on_press for the Clear Button (line 197) and the Color Button (line 206) and on_release for the Remove Button (line 200). The on_state event is specific to the ToggleButton class. This event is triggered every time the state of ToogleButton changes from 'normal' to 'down' and vice versa. The on_state is used in lines 203 and 209. All the events are bound to methods in the root which are defined in the generaloptions.py file:

```
210. # File name: generaloptions.py
211. from kivy.uix.boxlayout import BoxLayout
212. from kivy.properties import NumericProperty, ListProperty
213.
214. class GeneralOptions(BoxLayout):
215.     group_mode = False
216.     translation = ListProperty(None)
217.
218.     def clear(self, instance):
219.         self.drawing_space.clear_widgets()
220.
221.     def remove(self, instance):
222.         ds = self.drawing_space
223.         if len(ds.children) > 0:
224.             ds.remove_widget(ds.children[0])
225.
226.     def group(self, instance, value):
227.         if value == 'down':
228.             self.group_mode = True
229.         else:
230.             self.group_mode = False
231.             self.unselect_all()
232.
233.     def color(self, instance):
234.         pass
235.
236.     def gestures(self, instance, value):
237.         pass
238.
239.     def unselect_all(self):
240.         for child in self.drawing_space.children:
241.             child.unselect()
242.
243.     def on_translation(self,instance,value):
244.         for child in self.drawing_space.children:
245.             if child.selected:
246.                 child.translate(*self.translation)
```

The GeneralOptions methods are quite straightforward but they illustrate some other methods of `Widget`. The `clear` method removes all the widgets from the `DrawingSpace` instance through the `clear_widgets` method (line 219). The `remove` method removes the last added `Widget` instance accessing the children list (line 224). The `state` method modifies the `group_mode` attribute of line 215 according to the `'down'` or `'normal'` `ToggleButton` state. It also calls on the `unselect_all` method as defined in line 239.

Because of the group mode, the user is able to select several `DraggableWidgets` at the same time and drag them. The `unselect_all` method traverse over the list of children calling the internal method `unselect` of each `DraggableWidget` (line 79). The `color` and `gestures` methods will be completed in *Chapter 4*, *Improving the User Experience*.

Lastly, the `on_translation` method also traverse the children list on the internal `translate` method (line 71) of each `DraggableWidget`. The question is who calls the `on_translation` method? One of the most useful features of Kivy provides the answer to this question: this will be explained in the next section.

Creating your own events – the magical properties

This section covers the use of Kivy properties. A Kivy property triggers an event every time we modify it. There are different types of properties, from the simple `NumericProperty` or `StringProperty` to much more complex versions like `ListProperty`, `DictProperty`, or `ObjectProperty`. For example, if we define a `StringProperty`, called `text`, then an `on_text` event is going to be triggered each time text is modified.
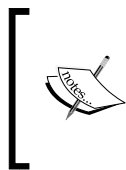
> The `on_translation` method (line 243) is associated with the `ListProperty` in line 216 called `translation`. Once we define a Kivy property, Kivy internally creates an event associated with that property. The property name is generated adding the prefix `on_` to the name of the property.

All the properties work in the same way. For example, the `state` property of `ToogleButton` is actually a property which creates the `on_state` event. We already used this event in line 203. We define the property and Kivy creates the event for us.

> In the context of this book, a property will always refer to a Kivy property and it should not be confused with a Python property, which is a different concept. An attribute is used to describe variables (references, objects, instances) that belong to the class. A property is always an attribute but an attribute is not necessarily a property.

In this section, we implement the `group_mode`. The group mode offers the user the possibility of selecting and dragging several figures (`DraggableWidgets` instances) at the same time by pressing the **Group** Button (line 201).

In order to do this, we can take advantage of the relation between the `translation` property and the `on_translation` method. Basically, every time we modify the `translation` property then the `on_translation` event is triggered. Suppose that we are dragging three figures at the same time (with the group mode) as shown in following screenshot:

 The three figures are selected, but the events are handled by the circle, since it is the one that has the pointer on top. The circle needs to tell the line and the stickman to translate. Instead of calling on the `on_translation` method, it only needs to modify the `translation` property and the `on_translation` method is called on. Let's include these changes in the `comicwidgets.py`. We need four modifications.

First, we need to add the `touched` attribute (line 249) to indicate which of the selected figures is receiving the event (for example, the circle in the previous screenshot). We do this in the constructor:

```
247. def __init__(self,  **kwargs):
248.     self.selected = None
249.     self.touched = False
250.     super(DraggableWidget, self).__init__(**kwargs)
```

Second, we need to set the `touched` attribute to `True` (Line 253) when one of the `DraggableWidget` instances receives the event. We do this in the `on_touch_down` method:

```
251. def on_touch_down(self, touch):
252.     if self.collide_point(touch.x, touch.y):
253.         self.touched = True
254.         self.select()
255.         return True
256.     return super(DraggableWidget, self).on_touch_down(touch)
```

 Third, we need to check that the `DraggableWidget` is the one that is currently being touched (received the `on_touch_down` event previously). We add this to the condition in line 259.

The most important change is the line 261. Instead of calling on the `translate` method directly, we modify the `translation` property of general options (`self.parent.general_options`) setting on the property the number of pixels the widget has been translated. This will trigger the `on_translation` method of `GeneralOptions` that at the same time calls the `translate` method for each selected `DraggableWidget`. The following is the code for the `on_touch_move`:

```
257. def on_touch_move(self, touch):
258.     (x,y) = self.parent.to_parent(touch.x, touch.y)
259.     if self.selected and self.touched and self.parent.
collide_point(x - self.width/2, y -self.height/2):
260.         go = self.parent.general_options
261.         go.translation=(touch.x-self.ix,touch.y-self.iy)
262.         return True
263.     return super(DraggableWidget, self).on_touch_move(touch)
```

Fourth, we need to set the `touched` attribute to `False` (line 265) on the `on_touch_up` event and also avoid calling the `unselect` method when we are using the `group_mode` (line 267). Here is the code for the `on_touch_up` method:

```
264. def on_touch_up(self, touch):
265.     self.touched = False
266.     if self.selected:
267.         if not self.parent.general_options.group_mode:
268.             self.unselect()
269.     return super(DraggableWidget, self).on_touch_up(touch)
```

This example could be considered artificial since we theoretically could have called on the `on_translation` method from the start. In order to achieve consistency of the internal state of a variable and the screen display of it, however, properties are crucial. The example from the next section should improve understanding of this.

Kivy and properties

Even though we have only touched on explanations of properties in the section, the truth is that we have been using them since the beginning of this chapter.Kivy's internals are full of properties. They are almost everywhere. For example, when we implemented `DraggableWidget`, we simply modified the property `center_x` and `center_y` (line 72) and the whole `Widget` was updated because there are properties involved in the use of `center_x`.

The last example in this chapter illustrates how powerful the Kivy properties are. Here is the code for `statusbar.py`:

```
270. # File name: statusbar.py
271. from kivy.uix.boxlayout import BoxLayout
272. from kivy.properties import NumericProperty, ObjectProperty
273.
274. class StatusBar(BoxLayout):
275.     counter = NumericProperty(0)
276.     previous_counter = 0
277.
278.     def on_counter(self, instance, value):
279.         if value == 0:
280.             self.msg_text = "Drawing space cleared"
281.         elif value - 1 == self.__class__.previous_counter:
282.             self.msg_text = "Widget added"
283.         elif value + 1 == self.previous_counter:
284.             StatusBar.msg_text = "Widget removed"
285.         self.previous_counter = value
```

Sometimes, the way Kivy properties work can be perceived as confusing by some advanced Python or Java programmers. In order to disentangle the problem, remember that we need to distinguish between the static attribute of a class, and the attribute of a class instance.

> In Python, the `previous_counter` (line 276) is a static attribute of the `StatusBar` class. This means that it is shared among all the `StatusBar` instances, and it can be accessed in any of the ways shown in lines 281, 283, and 285 (however line 281 is recommended). In contrast, the `selected` variable (line 248) is an attribute of a `DraggableWidget` instance. This means that there is a `selected` variable per `StatusBar` object. It is not shared among them. They are created until the constructor (`__init__`) is called on. The only way to access it is through `obj.selected` (line 248).

The confusion happens when a programmer assumes that the `counter` is a static attribute of the `StatusBar` class, because `counter` is defined in an equivalent way of the Python static attributes (for example, the `previous_counter`). The assumption is incorrect.

> Kivy properties are declared as static attribute classes (so as if they belong to the class) but they are always internally "transformed" to attribute instances. They actually belong to the object as if we would have declared them in the constructor.

After this being clarified, we can move on to study the example. The `counter` is defined as a `NumericProperty` in line 275. Its corresponding `on_counter` method (line 278) modifies a `Label` (`msg_text`) defined in the `statusbar.kv` file:

```
286. # File name: statusbar.kv
287. #:import statusbar statusbar
288. <StatusBar>:
289.     msg_text: _msg_label.text
290.     orientation: 'horizontal'
291.     Label:
292.         text: 'Total Figures: ' + str(root.counter)
293.     Label:
294.         id: _msg_label
295.         text: "Kivy started"
```

Note that we are using an `id` (line 294) again in order to define `msg_text` (line 289). Also, we are using the `counter` defined in line 275 to update the **Total Figures** message in line 292. The specific part (`str(root.counter)`) of `text` is updated automatically when the `counter` is modified.

So, we just need to modify `counter`. This is done in `drawingspace.py`:

```
296. # File name: drawingspace.py
297. from kivy.uix.relativelayout import RelativeLayout
298.
299. class DrawingSpace(RelativeLayout):
300.     def on_children(self, instance, value):
301.         self.status_bar.counter = len(self.children)
```

We have updated the `counter` with the length of the `DrawingSpace`'s `children` in the method `on_children`. Then, `on_children` is called on every time we add (line 108 or 133) or remove (line 219 or 224) widgets from the `children` list of the `DrawingSpace` because `children` is also a Kivy property.

Don't forget to import this file into `drawingspace.py` in the `drawingspace.kv` file:

```
302. # File name: drawingspace.kv
303. #:import drawingspace drawingspace
304. <DrawingSpace@RelativeLayout>:
```

The following diagram shows the chain of elements (properties, methods, and widgets) that are associated with the `children` property:
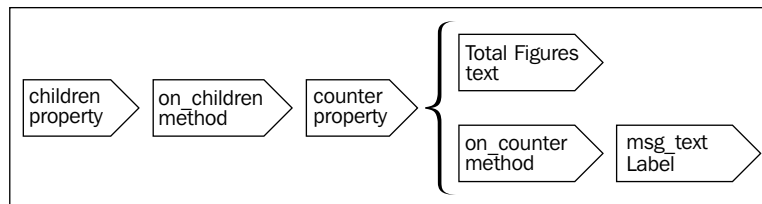
It is important to compare again the way we have to gain access to the `counter` property and the `msg_label` attribute. We define the `counter` property in the `StatusBar` (line 275) and use it in the `Label` through the `root` (line 292). In the `msg_label` case, we started defining the `id` (line 294) and then the attribute of the Kivy language (line 289). We could also make the `msg_label` an `ObjectProperty` if we for example, define something like `msg_label = ObjectProperty( None)` in the header `StatusBar`.

> Remember that an attribute is not necessarily a Kivy property. An attribute is an element of the class whereas a Kivy property associates the attribute with an event.

You can find the complete list of available properties in the Kivy API (`http://kivy.org/docs/api-kivy.properties.html`). There are two specific properties that, at least, should be mentioned: `BoundedNumericProperty` and `AliasProperty`. The `BoundedNumericProperty` property allows setting maximum and minimum values. If the value is beyond the range, an `Exception` is thrown. The `AliasProperty` property allows us to create our own properties in the case that the necessary property does not exist.

One last thing that deserves attention is that attributes of the vertex instructions are used as properties when we create them with Kivy language. For example, if we change the position of the line inside the `ToolLine`, it will be updated automatically. However, this just applies inside the Kivy language, not when we add the vertex instructions dynamically, as we did in `toolbox.py`. In our case, we had to remove and create a new vertex instruction every time we needed to update the figures (line 119 to 121). However, we could have created our own properties to handle the updates.

Summary

We have covered most of the topics related to event handling in this chapter. You have learned how to override different kind of events, dynamic binding and unbinding, assigning events in the Kivy language, and creating our own. You also learned about Kivy properties, how to manage the localization of coordinates to different widgets, and many methods related to adding, removing, and updating objects of the `Kivy Widget` and `canvas`. Here are the events, methods, properties, and attributes that were covered:

- The events: `on_touch_up`, `on_touch_move` and `on_touch_down` (of `Widget`); `on_press` and `on_release` (of `Button`) and `on_state` (of `ToggleButton`)

- The attributes `x` and `y` of a `MotionEvent` (`touch`); `center_x`, `center_y`, `canvas`, `parent`, `children` of `Widget` and `state` of `ToggleButton`.

- The following methods of `Widget`:
    - `bind` and `unbind` to attach events dynamically
    - `collide_points`, `to_parent`, `to_local`, `to_window`, and `to_widget` to work with coordinates
    - `add_widget`, `remove_widget`, and `clear_widgets` to dynamically modify the children widgets

- The methods `add` and `remove` of canvas to dynamically add and remove vertex and context instructions

- Kivy properties: `NumericProperty` and `ListProperty`

In general, all the properties that we have used with the Kivy language are accessible from Python. For example, we accessed the `text` of a `Label` but we could also have accessed the `size` or `pos` of a `Widget`. However, we cannot access all the properties of the instructions. For example, the `ellipse` of line is protected. Also, the `MotionEvent` (`touch` event) has many more attributes than the `x` and `y` used in this chapter. For example, type of touch, the number of taps (or clicks), duration, the input device, and many more that we can use for advanced tasks (`http://kivy.org/docs/api-kivy.input.motionevent.html#kivy.input.motionevent.MotionEvent`).

Finally, there were two other important types of events related to the clock and the keyboard. This chapter was focused on `Widget` and property events but we will learn how to use other events in *Chapter 5*, *Invaders Revenge – An Interactive Multi-touch Game*. The next chapter is going to introduce a list of interesting topics of Kivy in order to improve the user experience with our Comic Creator.

4
Improving the User Experience

This chapter gives and overview over selected components that Kivy provides to make the programmer's life easier when it is time to improve the user experience. Most of them are related to specific widgets that already include the functionality; in this case, you are going to learn the basic techniques to control them. Others facilitate a deeper comprehension of the drawing context of Kivy or the use of classes to handle gestures.

The following are the knowledge you will acquire in the chapter:

- Switching between different screens
- Using pre-build complex widgets to select colors
- Controlling the visible area of the canvas
- Rotate and scale with multi-touch gestures
- Creating single gestures to draw on the screen

More importantly, we will learn how to incorporate them into a current working project. This will reinforce your previously acquired knowledge and understanding of how Kivy works. At the end of this chapter, you should feel comfortable to face possible problems that you may find developing with Kivy. Moreover, we should be able to dig into the Kivy API and quickly understand the use of different components and widgets.

Screen manager – selecting colors for the figures

The ScreenManager class lets us handle different screens in the same window. In Kivy, screens are preferred over windows because we are programming for different devices with different screen sizes. Therefore, it is difficult (if not impossible) to have windows that adapt properly to all devices.

So far, all our figures have been of the same color and that is boring. So, let us allow the user to add some color to make the Comic Creator more fun. Kivy provides us with a Widget called ColorPicker, which is displayed in the following screenshot:
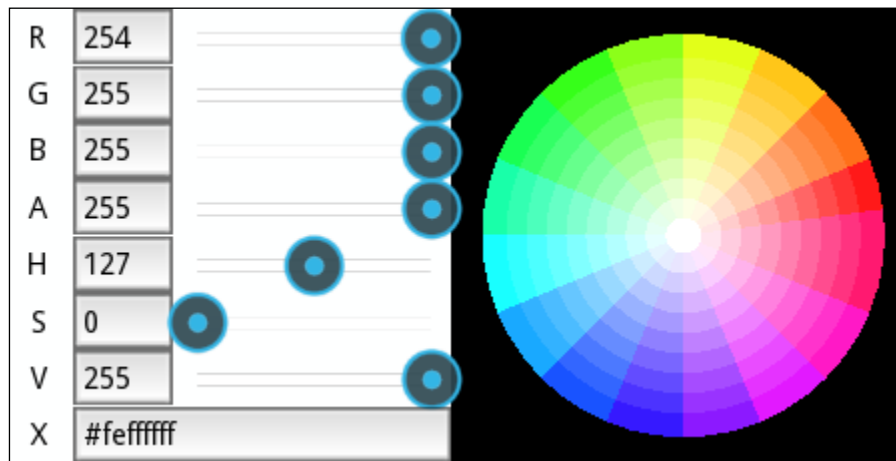
Kivy Color Picker

As you can see, this Widget requires a wide space, so it is difficult to accommodate it in our current interface. Let's use a ScreenManager instance to solve this problem. This basically allows us to have multiple screens instead of just one Widget (ComicCreator) and switch easily between them. The following is a new Kivy file (comicscreenmanager.kv) that contains the ComicScreenManager class definition:

```
1. # File name: comicscreenmanager.kv
2. #:import FadeTransition kivy.uix.screenmanager.FadeTransition
3. <ComicScreenManager>:
4.     transition: FadeTransition()
5.     color_picker: _color_picker
6.     ComicCreator:
7.     Screen:
8.         name: 'colorscreen'
9.         ColorPicker:
10.            id: _color_picker
```

```
11.              Button:
12.                  text: "Select"
13.                  pos_hint: {'center_x': .75, 'y': .05}
14.                  size_hint: None, None
15.                  size: 150, 50
16.                  on_press: root.current = 'comicscreen'
```

In line 5, there is an attribute `color_picker` that points to our instance of `ColorPicker` in line 9. However, notice that we haven't added a `ColorPicker` instance directly to `ComicScreenManager`. Instead we have embedded the `ColorPicker` instance inside a `Screen` `Widget`. In line 6, we added an instance of our `ComicCreator`.

> A `ScreenManager` instance must contain the widgets of the `Screen` base class. No other types of Widget (`Label`, `Button` or `layouts`) are allowed.

We need to change our `ComicCreator`, so it inherits `Screen`. We do this in the `comiccreator.kv` header:

```
17. # File name: comiccreator.kv
18. <ComicCreator@Screen>:
19.     name: 'comicscreen'
20.     AnchorLayout:…
```

We assigned the `name` `'comicscreen'` to identify the screen (line 19). This `name` is used in the `Button` instance that we added to the `ColorPicker` (line 11). In line 16, we bind the method `on_press` with the Python code `root.current = 'comicscreen'`. Please note we are adding Python code directly instead of calling on a method as we did in *Chapter 3*, *Widget Events – Binding Actions*. In this case, the `root` is actually the `ScreenManager` class and the `current` property tells it which the active `Screen` is. The value `'comicscreen'` is the `name` as just explained.

We indentify the `ColorPicker` `Screen` instance with the `name` `'colorscreen'` on line 8. This `name` is used to activate the `ColorPicker` in the GeneralOptions area. We need to modify the `color` method of `generaloptions.py`:

```
21. def color(self, instance):
22.     self.comic_creator.manager.current = 'colorscreen'
```

The **Color** `Button` now switches the `Screen` in order to display the `ColorPicker`. Remember that the `GeneralOptions` class has a reference (`comic_creator`) to the `ComicCreator` instance. Since `ComicCreator` is a `Screen`, it can access its corresponding `ScreenManager` through the `manager` attribute. Therefore, it also can change the `current` `Screen`, analogous to line 16.

The `ComicScreenManager` instance becomes the main `Widget` of the comic creator project so the `comicreator.py` file has to change accordingly:

```
23. # File name: comiccreator.py
24. from kivy.app import App
25. from kivy.lang import Builder
26. from kivy.uix.screenmanager import ScreenManager
27.
28. Builder.load_file('toolbox.kv')
29. Builder.load_file('comicwidgets.kv')
30. Builder.load_file('drawingspace.kv')
31. Builder.load_file('generaloptions.kv')
32. Builder.load_file('statusbar.kv')
33. Builder.load_file('comiccreator.kv')
34.
35. class ComicScreenManager(ScreenManager):
36.     pass
37.
38. class ComicScreenManagerApp(App):
39.     def build(self):
40.         return ComicScreenManager()
41.
42. if __name__=="__main__":
43.     ComicScreenManagerApp().run()
```

Since we have changed the name of the `App` to `ComicScreenManagerApp` (line 43), we are explicitly loading the `comiccreator.kv` file (line 33). One last interesting thing about the `ScreenManager` is that we can use transitions. Just as an example, the lines 2 and 4 import and use a simple `FadeTransition`.

> Kivy provides a set of transitions (`FadeTransition`, `SwapTransition`, `SlideTransition`, and `WipeTransition`) for switching between the `Screen` instances of a `ScreenManager`. Check the Kivy API for more information on how to customize them with different parameters at `http://kivy.org/docs/api-kivy.uix.screenmanager.html`.

After these changes, we can switch between the two screens: `ColorPicker` and `ComicCreator`. However, the selection of the color still has no effect on the drawing process. The next section covers how to add the color to the figures we draw.

Color Control on the canvas – coloring figures

The previous section focused on the selection of colors. Now, we are going to actually use the selected color. Assigning a color can be tricky if we are not careful. The main issue is that there isn't a `color` property neither for the widgets nor for the graphics (`Line`, `Rectangle`) inside the canvas. If you recall, in *Chapter 3, Widget Events – Binding Actions*, `Color` is a context instruction that we must add to the canvas.

Moreover, we have to be sure that we add the instruction before we draw the actual figure. Something like `PushMatrix` and `PopMatrix` could be the solution but they only apply for transform instructions (`Translate`, `Rotate`, and `Scale`).

Another way to do it could be to add a `color` property to the `DraggableWidget` and then have a temporary variable that keeps the previous `color`. After we draw, we use the temporary variable to reset the previous color. That is quite a hassle for something that should be simple.

It is such a hassle that Kivy doesn't worry too much about it. Kivy ensures that the elements inside any widget are painted as the corresponding color, but doesn't care about cleaning the context state. Let us study a small example (out of the Comic Creator project) to understand this concept:

```
44. # File name: color.py
45. from kivy.app import App
46. from kivy.uix.gridlayout import GridLayout
47. from kivy.lang import Builder
48.
49. Builder.load_string("""
50. <GridLayout>:
51.     cols:2
52.     Label:
53.         color: 1,0,0,1
54.         canvas:
55.             Line:
56.                 points: self.x, self.y, self.x + self.width,
    self.y + self.height
57.     Widget:
58.         canvas:
59.             Line:
60.                 points: self.x, self.y, self.x +self.width,
    self.y + self.height
61. """)
62.
63. class LabelApp(App):
```

```
64.    def build(self):
65.        return GridLayout()
66.
67. if __name__=="__main__":
68.    LabelApp().run()
```

> Notice that we use the `load_string` method of the `Builder` class instead of `load_file`. This method allows us to embed Kivy language statements inside a Python code file.

One of the properties of `Label` is called `color`. It changes the color of the `Label` text. We change this `color` to red (line 53) on the first `Label` but doesn't clean the context. Observe the result in the following screenshot:

Color property of Label

Not only the `Line` of the `Label` (added first) but also the `Line` of the `Widget` (added second), so when the context has changed, it stays changed.

"When in Rome, do as Romans do". Kivy tries to keep all its components as simple as possible to avoid overloading instructions. We will follow this approach for the colors and just worry about the color each time we need to use it, and the rest of the components should care of their own color. This is of course not the only solution, but it is simple and consistent with Kivy.

We can now implement the changes. There are only three methods where we draw in the drawing space (all of them are in the `toolbox.py` file). Here are those methods with the corresponding new lines highlighted:

- Method `draw` in `ToolStickman`:

```
69. def draw(self, ds, x, y):
70.     sm = StickMan(width=48, height=48)
71.     sm.center = (x,y)
72.     screen_manager = self.parent.comic_creator.manager
73.     color_picker = screen_manager.color_picker
74.     sm.canvas.before.add(Color(*color_picker.color))
75.     ds.add_widget(sm)
```

- Method `draw` (class `ToolFigure`):

```
76. def draw(self, ds, x, y):
77.     (self.ix, self.iy) = (x,y)
```

```
78.    screen_manager = self.parent.comic_creator.manager
79.    color_picker = screen_manager.color_picker
80.    with ds.canvas:
81.        Color(*color_picker.color)
82.        self.figure=self.create_figure(x,y,x+1,y+1)
83.    ds.bind(on_touch_move=self.update_figure)
84.    ds.bind(on_touch_up=self.end_figure)
```

- Method `widgetize` (class `ToolFigure`):

```
85. def widgetize(self,ds,ix,iy,fx,fy):
86.     widget = self.create_widget(ix,iy,fx,fy)
87.     (ix,iy) = widget.to_local(ix,iy,relative=True)
88.     (fx,fy) = widget.to_local(fx,fy,relative=True)
89.     screen_manager = self.parent.comic_creator.manager
90.     color_picker = screen_manager.color_picker
91.     widget.canvas.add(Color(*color_picker.color))
92.     widget.canvas.add(self.create_figure(ix,iy,fx,fy))
93.     ds.add_widget(widget)
```

All three methods have a pair of specific instructions in common; you can find them in lines 72 and 73, 78 and 79, 89 and 90. These are reference chains to get access to the `ColorPicker` instance. After that we just add a `Color` instruction to the canvas (as we learned in *Chapter 2, Graphics – The Canvas*) using the selected `color` in the `color_picker`.

We also use `canvas.before` in the `draw` method of the `ToolStickman` class (line 74). This is used to ensure that the `Color` instruction is executed before the instructions we added in the `canvas` of the `Stickman` (`comicwidgets.kv` file). This is not necessary in the other two methods because we have full control of the canvas order inside those methods.

Lastly, please don't forget to import the `Color` class in the header of the file: `from kivy.graphics import Line, Color`. We can now take a break, and enjoy the result of our hard work with our Comic Creator:
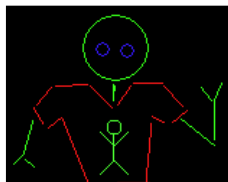
At a later point in time, we can discuss whether our drawing is just an avid Comic Creator fan or a narcissistic alien. For now, it seems more useful that you learn how to limit the drawing space to the specific area that occupies in the window.

StencilView – limiting the drawing space

In *Chapter 3*, *Widget Events – Binding Actions*, we avoided drawing outside of the drawing space by using simple mathematics and `collide_points`. It was far from perfect (for example, it fails in the group mode or when we are resizing) and it was tedious and prone to programming mistakes.

`StencilView` is the way to go here. With a **StencilView** instance , you can avoid drawing outside of the area that is defined by it. First, let's modify the file `drawingspace.py` with the following header:

```
94. # File name: drawingspace.py
95. from kivy.uix.stencilview import StencilView
96.
97. class DrawingSpace(StencilView):
98.     ...
```

We have substituted `RelativeLayout` with `StencilView`. The `StencilView` class doesn't use relative coordinates (as the `RelativeLayout` class does) but we would like to keep that behavior in the drawing space. We can fix this problem modifying the top-right `AnchorLayout`, so the `DrawingSpace` instance is inside a `RelativeLayout` instance. We do this in `comiccreator.kv`:

```
99.     AnchorLayout:
100.        anchor_x: 'right'
101.        anchor_y: 'top'
102.        RelativeLayout:
103.            size_hint: None,None
104.            width: root.width - _tool_box.width
105.            height: root.height -
     _general_options.height - _status_bar.height
106.            DrawingSpace:
107.                id: _drawing_space
108.                general_options: _general_options
109.                tool_box: _tool_box
110.                status_bar: _status_bar
```

The `DrawingSpace` instance (line 108) is now embedded inside a `RelativeLayout` instance (line 104) keeping its id (line 109) and attributes (lines 110 to 112). Since we have a new level of indentation and the `DrawingSpace` class is not relative itself, this affects the way we are localizing the coordinates in the `ToolBox` instance, specifically, in `update_figure` and `end_figure` of the class `ToolFigure`. Following is the new code for those methods in `toolbox.py`:

```
111. def update_figure(self, ds, touch):
112.     ds.canvas.remove(self.figure)
```

```
113.    with ds.canvas:
114.            self.figure = self.create_figure(self.ix,
     self.iy,touch.x,touch.y)
115.
116. def end_figure(self, ds, touch):
117.        ds.unbind(on_touch_move=self.update_figure)
118.        ds.unbind(on_touch_up=self.end_figure)
119.        ds.canvas.remove(self.figure)
120.        self.widgetize(ds,self.ix,self.iy,touch.x,touch.y)
```

We have removed some instructions. First off, we don't need to use the `to_widget` method anymore since we are already getting the coordinates from the `RelativeLayout` parent. And secondly, we don't need to worry about `collide_points` in the `update_figure` method because `StencilView` will be in charge of it.

With just a few changes we have ensured that nothing will be drawn outside of the drawing space. Please note that by default (`size_hint: 1, 1`) the `DrawingSpace` instance occupies all the area of the `RelativeLayout` parent. The next section will teach you how to drag, rotate, and scale the figures.

Scatter – multitouching to drag, rotate, and scale

In the previous chapter you learned how to use events to drag widgets. You learned how to use the `on_touch_up`, `on_touch_move` and `on_touch_down` events. However, the `Scatter` class already provides that functionality and also lets us scale and rotate using two fingers. All the functionality is included inside the `Scatter` class, however, we need to apply a few changes to keep our project consistent. In particular, we still want our group mode to work, so that translating, scaling, and rotating can be happening at the same time. Let us implement the changes in four big steps in the `comicwidgets.py` file:

1. Substitute in the `DraggableWidget` base class. Let's use `Scatter` instead of `RelativeLayout` (line 122 and 125):

    ```
    121. # File name: comicwidgets.py
    122. from kivy.uix.scatter import Scatter
    123. from kivy.graphics import Line
    124.
    125. class DraggableWidget(Scatter):
    ```

    > Both, `Scatter` and `RelativeLayout` use relative coordinates.

2. Make sure that the `on_touch_down` event is propagated to the base class (`Scatter`) by calling the `super` method (line 130) before `return True` (line 131) inside the condition. If we don't do that the `Scatter` will never receive the event `on_touch_down` and nothing will happen:

```
126. def on_touch_down(self, touch):
127.     if self.collide_point(touch.x, touch.y):
128.         self.touched = True
129.         self.select()
130.         super(DraggableWidget, self).on_touch_down(touch)
131.         return True
132.     return super(DraggableWidget, self).on_touch_down(touch)
```

> The super method is for the base class (`Scatter`), the `return` method for the parent (`DrawingSpace`)

3. Remove the `on_touch_move` method and add an `on_pos` method. Since the `Scatter` will be responsible for the dragging we don't need `on_touch_move` anymore. Instead we are going to use the `pos` property that is modified by the `Scatter`. Remember that the properties trigger an event that will call the `on_pos` method:

```
133. def on_pos(self, instance, value):
134.     if self.selected and self.touched:
135.         go = self.parent.general_options
136.         go.translation = (self.center_x- self.ix,
    self.center_y - self.iy)
137.         self.ix = self.center_x
138.         self.iy = self.center_y
```

4. `Scatter` has other two properties: `rotation` and `scale`. So, we can use the same idea of `pos` and `on_pos`, and add the `.on_rotation` and `on_scale` methods.

```
139.     def on_rotation(self, instance, value):
140.         if self.selected and self.touched:
141.             go = self.parent.general_options
142.             go.rotation = value
143.
144.     def on_scale(self, instance, value):
145.         if self.selected and self.touched:
146.             go = self.parent.general_options
147.             go.scale = value
```

5. The `on_rotation` and `on_scale` methods simply modify a couple of new properties we added to General Options that will help us to keep the group mode working. Following is the header of `generaloptions.py`:

```
148. # File name: generaloptions.py
149. from kivy.uix.boxlayout import BoxLayout
150. from kivy.properties import NumericProperty, ListProperty
151.
152. class GeneralOptions(BoxLayout):
153.     group_mode = False
154.     translation = ListProperty(None)
155.     rotation = NumericProperty(0)
156.     scale = NumericProperty(0
```

6. Next, we are importing `NumericProperty` together with `ListProperty` (line 143); and we are creating the two missing properties: `rotation` and `scale` (lines 148 and 149). We are also adding the `on_rotation` (line 150) and `on_scale` (line 155) methods associated with the `rotation` and `scale` properties), which will ensure that all the `selected` components are rotated or scaled at once:

```
157.     def on_rotation(self, instance, value):
158.         for child in self.drawing_space.children:
159.             if child.selected and not child.touched:
160.                 child.rotation = value
161.
162.     def on_scale(self, instance, value):
163.         for child in self.drawing_space.children:
164.             if child.selected and not child.touched:
165.                 child.scale = value
```

A final modification is necessary. We have changed the `on_translation` method to check that the current `child` in the loop is not the one being `touched`. This way, we avoid an infinite recursion (with the modification of `pos` inside the `translate` method). Here is the new `on_translation` method on `generaloptions.py`:

```
166.     def on_translation(self,instance,value):
167.         for child in self.drawing_space.children:
168.             if child.selected and not child.touched:
169.                 child.translate(*self.translation)
```

At this point, we are able to translate, rotate, or scale the figures with our fingers, even in the group mode.

> Kivy provides a way to simulate multitouch with the mouse. It is limited, but you can still test this section with your one-mouse laptop. All you have to do is right click on the figure you want to rotate. A translucent red circle will appear on the screen. Then you can use the normal left dragging as if it were a second finger to rotate or scale.

The next screenshot cut shows our `StickMan`, being rotated and scaled at the same time as the line next to him. The small `StickMan` on the right is just a reference for the original size. The simulated multitouch gesture is being applied to the line on the right and that is why you can see a red dot:
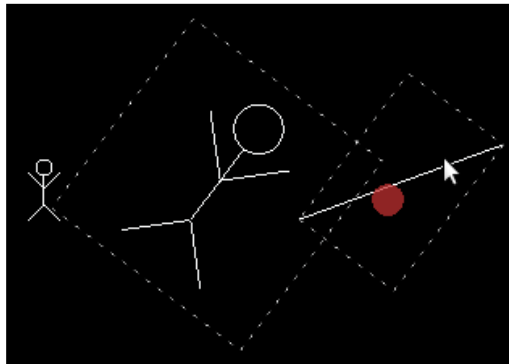
In *Chapter 1*, *GUI Basics: Building an Interface*, we postponed `ScatterLayout` explanation to this chapter. Here it is:

> `ScatterLayout` is a Kivy layout that inherits from `Scatter` and contains `FloatLayout`. This allows you to use the `size_hint` and `pos_hint` properties when you add widgets inside to it. `ScatterLayout` also uses relative coordinates.

That doesn't mean you cannot add other widgets inside a simple `Scatter`, it just means that `Scatter` doesn't honor `size_hint` or `pos_hint`.

Recording gestures – line, circles, and cross

What about drawing with one finger? Can we recognize gestures? It is possible to do this with Kivy, but first we need to record the gestures that we want to use. A gesture is represented as a long string that contains the points of a stroke over the screen. The following code is not part of the project and can be run with `python gesturerecorder.py`:

```
170. # File Name: gesturerecorder.py
171. from kivy.app import App
172. from kivy.uix.floatlayout import FloatLayout
173. from kivy.graphics import Line, Ellipse
174. from kivy.gesture import Gesture, GestureDatabase
175.
176. class GestureRecorder(FloatLayout):
177.
178.     def on_touch_down(self, touch):
179.         self.points = [touch.pos]
180.         with self.canvas:
181.             Ellipse(pos=(touch.x-5,touch.y-5),size=(10,10))
182.             self.line = Line(points=(touch.x, touch.y))
183.
184.     def on_touch_move(self, touch):
185.         self.points += [touch.pos]
186.         self.line.points += [touch.x, touch.y]
187.
188.     def on_touch_up(self, touch):
189.         self.points += [touch.pos]
190.         gesture = Gesture()
191.         gesture.add_stroke(self.points)
192.         gesture.normalize()
193.         gdb = GestureDatabase()
194.         print "Gesture:", gdb.gesture_to_str(gesture)
195.
196. class GestureRecorderApp(App):
197.     def build(self):
198.         return GestureRecorder()
199.
200. if __name__=="__main__":
201.     GestureRecorderApp().run()
```

The previous code prints the gesture string representations using the `Gesture` and `GestureDatabase` classes (line 174). The `on_touch_down`, `on_touch_move`, and `on_touch_up` methods collects the `points` of the stroke lines 179, 185 and 189. The following screenshot are examples of strokes collected with `gesturerecorded.py`:
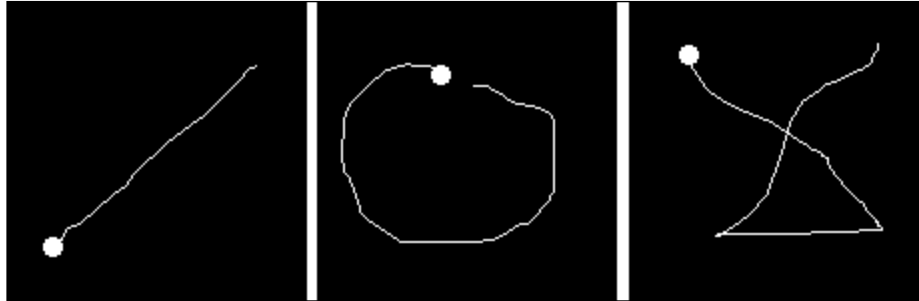
The small `Circle` in the preceding figures (lines 180 and 181) indicates the starting point and the line indicates the path that the stroke follows. The most relevant part is coded in lines 190 to 194. We create a `Gesture` (line 190), add the `points` for the stroke (line 191), `normalize` to a default number of points (line 192) and create a `GestureDatabase` instance (line 193) that we use in line 194 to generate the string and print it on the screen.

The following screenshot shows the terminal output for the stroke line (corresponding to the first left figure in the preceding figures set):

In the preceding screenshot, the text output starting with `'eNq1Vktu...'` is the gesture string representation. We use those long strings as descriptors of the gestures that Kivy understands and uses to associate the stroke with any action we want to perform. The last section explains how to achieve this.

Simple gestures – drawing with the finger

The previous section explained how to obtain string representation from gestures. The current section explains how to use those string representations to recognize the gestures. We have copied the strings that were generated by the strokes in the previous section into a new file called `gestures.py`. The strings are assigned to different variables. The following code corresponds to `gestures.py`:

```
202. # File Name: gestures.py
203. line45_str = 'eNq1VktuI0cM3fdFrM0I...
204. circle_str = 'eNq1WMtuGzkQvM+P2JcI/Sb5A9rrA...
205. cross_str = 'eNq1V9tuIzcMfZ8fSV5qiH...
```

Only the first few characters of the strings are shown in the previous code because of space concern, but you can use the previous section to generate your own strings.

Next, we are going to use this string in the `drawingspace.py` file. Let's start with its header first:

```
206. # File name: drawingspace.py
207. from kivy.uix.stencilview import StencilView
208. from kivy.gesture import Gesture, GestureDatabase
209. from gestures import line45_str, circle_str, cross_str
210.
211. class DrawingSpace(StencilView):
```

In the preceding code we import the `Gesture` and `GestureDatabase` classes together with the gesture string representations added to `gestures.py.`. We have added several methods to the `DrawingSpace` class. Let's quickly review each of them and at the end we review the new parts:

- `__init__`: This method creates the attributes of the class and fills the `GestureDatabase`:

```
212. def __init__(self, *args, **kwargs):
213.     super(DrawingSpace, self).__init__()
214.     self.gdb = GestureDatabase()
215.     self.line45 = self.gdb.str_to_gesture(line45_str)
216.     self.circle = self.gdb.str_to_gesture(circle_str)
217.     self.cross = self.gdb.str_to_gesture(cross_str)
218.     self.line135 = self.line45.rotate(90)
219.     self.line225 = self.line45.rotate(180)
220.     self.line315 = self.line45.rotate(270)
221.     self.gdb.add_gesture(self.line45)
222.     self.gdb.add_gesture(self.line135)
223.     self.gdb.add_gesture(self.line225)
```

```
224.       self.gdb.add_gesture(self.line315)
225.       self.gdb.add_gesture(self.circle)
226.       self.gdb.add_gesture(self.cross)
```

- `activate` and `deactivate`: This method binds and unbinds the methods to the touch events in order to start the gesture recognition mode. These methods are called on by the gesture `Button` of the `GeneralOptions`:

```
227. def activate(self):
228.       self.bind(on_touch_down=self.down,
229.                 on_touch_move=self.move,
230.                 on_touch_up=self.up)
231.
232. def deactivate(self):
233.       self.unbind(on_touch_down=self.down,
234.                 on_touch_move=self.move,
235.                 on_touch_up=self.up)
```

```
down, move and up: These methods record the points of the stroke
in a very similar way to the previous section:
236. def down(self, ds, touch):
237.    if self.collide_point(*touch.pos):
238.         self.points = [touch.pos]
239.         self.ix = self.fx = touch.x
240.         self.iy = self.fy = touch.y
241.    return True
242.
243. def move(self, ds, touch):
244.      if self.collide_point(*touch.pos):
245.          self.points += [touch.pos]
246.          self.min_and_max(touch.x, touch.y)
247.      return True
248.
249. def up(self, ds, touch):
250.      if self.collide_point(*touch.pos):
251.          self.points += [touch.pos]
252.          self.min_and_max(touch.x, touch.y)
253.          gesture = self.gesturize()
254.          recognized = self.gdb.find(gesture, minscore=0.50)
255.          if recognized:
256.               self.discriminate(recognized)
257.      return True
```

- `gesturize`: This creates a `Gesture` instance from the collected points on the previous methods:

```
258. def gesturize(self):
259.     gesture = Gesture()
260.     gesture.add_stroke(self.points)
261.     gesture.normalize()
262.     return gesture
```

- `min_and_max`: This keeps track of the extreme points of the stroke:

```
263. def min_and_max(self, x, y):
264.     self.ix = min(self.ix, x)
265.     self.iy = min(self.iy, y)
266.     self.fx = max(self.fx, x)
267.     self.fy = max(self.fy, y)
```

- `discriminate`: This decides which method to call according to the recognized gesture:

```
def discriminate(self, recognized):
268.     if recognized[1] == self.cross:
269.         self.add_stickman()
270.     if recognized[1] == self.circle:
271.         self.add_circle()
272.     if recognized[1] == self.line45:
273.         self.add_line(self.ix,self.iy,self.fx,self.fy)
274.     if recognized[1] == self.line135:
275.         self.add_line(self.ix,self.fy,self.fx,self.iy)
276.     if recognized[1] == self.line225:
277.         self.add_line(self.fx,self.fy,self.ix,self.iy)
278.     if recognized[1] == self.line315:
279.         self.add_line(self.fx,self.iy,self.ix,self.fy)
```

- `add_circle`, `add_line`, `add_stickman`: These use the corresponding `ToolButton` of `ToolBox` to add a figure according to the recognized gesture:

```
281. def add_circle(self):
282.     cx = (self.ix + self.fx)/2.0
283.     cy = (self.iy + self.fy)/2.0
284.     self.tool_box.tool_circle.widgetize
    (self, cx, cy, self.fx, self.fy)
285.
286. def add_line(self,ix,iy,fx,fy):
287.     self.tool_box.tool_line.widgetize(self,ix,iy,fx,fy)
288.
```

```
289. def add_stickman(self):
290.     cx = (self.ix + self.fx)/2.0
291.     cy = (self.iy + self.fy)/2.0
292.     self.tool_box.tool_stickman.draw(self,cx,cy)
```

- `on_children`: This just keeps the counter of the **status bar** updated:

```
293. def on_children(self, instance, value):
294.     self.status_bar.counter = len(self.children)
```

The new elements are presented in the `__init__` (lines 212 to 226) and `up` (lines 249 to 257) methods. In the `__init__` method, we create the `GestureDatabase` instance (line 214) and use it to create the gestures from the strings (lines 215 to 217). We rotate 90° the `line 45` gesture (lines 218 to 220) gesture four times, so the `GestureDatabase` instance will recognize the gestures in different directions. Then, the method charges the `GestureDatabase` with the generated gestures (lines 221 to 226).

In the `up` method, we search into the `GestureDatabase` instance with the `find` method (line 254). The `minscore` parameter is used to indicate the precision of the search. We are using a low level since we know that the strokes are very different and that a mistake can be easily undone. The found gesture is kept in the `recognized` variable. This variable is a pair where the first element is the score of the recognition and the second value is the actual recognized gesture. Then, the method `discriminate` (lines 268 to 280) is called on. This method uses the recognized gesture information to decide which figure is drawn. In the case of the lines, it also decides the order in which to send the coordinates to match the direction.

One last change is required in the `generaloptions.py` file. That definition of the `gestures` method:

```
295. def gestures(self, instance, value):
296.     if value == 'down':
297.         self.drawing_space.activate()
298.     else:
299.         self.drawing_space.deactivate()
```

When the **Gestures** `ToggleButton` is in the `'down'` state, the GeneralOptions buttons continue working but the Tool Box is mostly frozen. The reason for this is the order in which we added the widgets to the screen in the `comiccreator.kv` file and the events triggering order. Luckily, this behavior is acceptable since we don't use those buttons to draw with the fingers.

Summary

This chapter has covered some specific and useful topics that improve the user experience. We have added several screens and switch between them with `ScreenManager`. You have learned how to use colors in the canvas and you should now have a good understanding of how this works internally. You also learned how to limit the drawing area to the drawing space with the `StencilView`. We used `Scatter` to add rotating and scaling capabilities to our `DraggablWidget` and expanded the functionality through the use of properties and associated events.

Finally, we introduced the use of gestures to make the interface more dynamic. Here is a review of all the classes with their respective methods, properties, and attributes that you have learned to use in this chapter:

- `ScreenManager`: The `transistion` and `current` properties
- The `FadeTransition`, `SwapTransition`, `SlideTransition`, and `WipeTransition` transitions
- `Screen`: The `name` and `manager` properties
- `ColorPicker`: The `color` property
- `StencilView`
- `Scatter`: The `rotate` and `scale` properties; and the `on_translate`, `on_rotate` and `on_scale` methods (events)
- `ScatterLayout`: The `size_hint` and `pos_hint` properties
- `Gesture`: The `add_stroke`, `normalize`, and `rotate` methods
- `GestureDatabase`: The `gesture_to_str`, `str_to_gesture`, `add_gesture`, and `find` methods

These are all useful components that help us to create more interactive and dynamic applications. Only the basic usage has been explained here, however, you can always check the Kivy API for a more comprehensive list of properties and methods.

The next chapter will introduce personalized multitouch-control, animations, as well as the clock and keyboard events. We are going to create a new interactive project, a game that resembles the arcade game Space Invaders.

5
Invaders Revenge – An Interactive Multitouch Game

This chapter introduces a collection of components and strategies to make animated and dynamic applications. Most of them are particularly useful for game development. This chapter is full of examples of how to combine the components and teaches strategies to control many events happening at the same time. The examples are all integrated in a completely new project, a version of the classic Space Invaders game. The following is a list of the main components that we will be working on within this chapter:

- **Atlas**: A Kivy package that allows us to loads images efficiently
- **Sound**: Classes that allows sound management
- **Animations**: Transitions, time control, events, and operations that can be applied to animate widgets.
- **Clock**: A class that allows us to schedule events
- **Multitouch**: A strategy that allows us to control different actions according to touches
- **Keyboard**: The Kivy strategy of capturing keyboard events

The first section presents an overview of the project, the GUI and game rules of the game. After that, we will follow a bottom-up approach. The simple classes that refer to individual components of the game will be explained, and additional topics of the chapter will then be introduced one after another. We will finish with the classes that have the main control over the game. By the end of this chapter, you should be able to start any game application you've always wanted to implement for your mobile device.

Invaders Revenge – an animated multitouch game

Invaders Revenge is the name of our Kivy version of Space Invaders©. The following screenshot shows you the game we are going to build in this chapter:
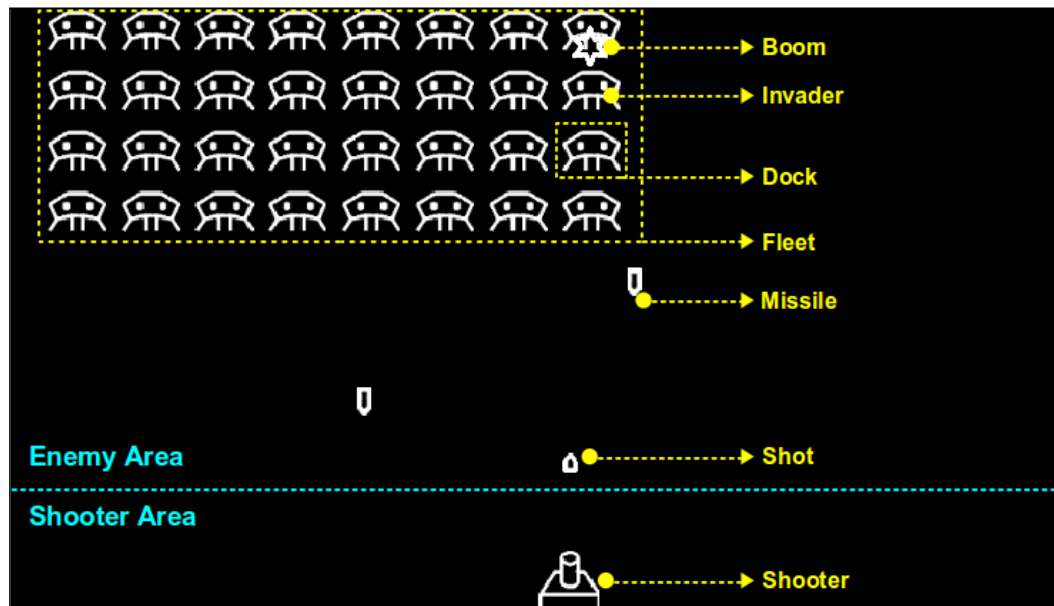
There are several markers in yellow and cyan color in the screenshot. They help identify the structure of our game; the game will consist of a Shooter (the player) which shoots (Shot) at a number of Invaders whose mission is to destroy the Shooter with their Missiles. The Invaders are organized in a Fleet (which moves horizontally) and sometimes an individual Invader can break out of the grid formation and fly around the screen before they go back to their corresponding Dock.

The cyan line across the screen indicates an internal division of the screen into the Enemy Area and the Shooter Area. This division is used to discriminate the actions that should occur according to touches that happen in different sections of the screen.

The skeleton of the game is presented in `invasion.kv`:

```
1. # File name: invasion.kv
2. <Invasion>:
3.    id: _invasion
4.    shooter: _shooter
5.    fleet: _fleet
```

```
6.    AnchorLayout:
7.      anchor_y: 'top'
8.      anchor_x: 'center'
9.      FloatLayout:
10.         id: _enemy_area
11.         size_hint: 1, .7
12.         Fleet:
13.           id: _fleet
14.           invasion: _invasion
15.           shooter: _shooter
16.           cols: 8
17.           spacing: 40
18.           size_hint: .5, .4
19.           pos_hint: {'top': .9}
20.           x: root.width/2-root.width/4
21.     AnchorLayout:
22.       anchor_y: 'bottom'
23.       anchor_x: 'center'
24.       FloatLayout:
25.         size_hint: 1, .3
26.         Shooter:
27.           id: _shooter
28.           invasion: _invasion
29.           enemy_area: _enemy_area
```

Basically, there are two `AnchorLayout` instances. The top one is the Enemy Area that contains the Fleet and the bottom one is the Shooter Area that contains the Shooter.
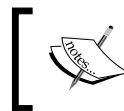
> Enemy Area and Shooter Area are very important for the logic of the game in order to distinguish the types of touches on the screen.

Atlas – efficient management of images

When it comes to applications that use many images, it is important to reduce their loading time, especially when they are requested from a remote server.
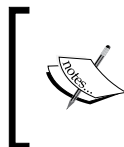
> One strategy to reduce the loading time is using an Atlas (also known as sprites). This reduces loading time because all the images are packed into one image, so it only needs to be requested once.

Here is the atlas image we use for the Invaders Revenge:

Instead of requesting five images for Invaders Revenge, we will just request the atlas image. We are also going to need a `json` file that tells us the exact coordinates of each image. The good news is that we don't need to do this manually. Kivy provides a simple command to create both the atlas image and the `json` file. Assuming that all the images are in a directory called `img`, we just need to open a terminal, go to the `img` directory and run the following command in the terminal:

```
python -m kivy.atlas invasion 100 *.png
```

> In order to execute the previous command you need to install the **Python Imaging Library** (**PIL**) from `https://pypi.python.org/pypi/PIL`.

The command contains three parameters, namely `basename`, `size`, and `images list`. The `basename` parameter is the prefix of the `json` (`invasion.json`) file and the atlas image (`invasion-0.png`). However, none of the images should be bigger than the size of the atlas. The next parameter is the `images list` that will be added to the atlas.

We employ the format `atlas://path/to/atlas/atlas_name/id` to use the atlas. The `id` file refers to the image file name without the extension. For example, normally we would have referenced the Shooter image like source: `'img/shooter.png'`. After generating the atlas, it becomes source: `'atlas://images/invasion/shooter'`. The following `image.kv` file presents the code for all the images of the Invaders Revenge:

```
30. # File name: images.kv
31. <Invader@Image>:
32.    source: 'atlas://img/invasion/invader'
33.    size_hint: None,None
34.    size: 40,40
35. <Shooter@Image>:
36.    source: 'atlas://img/invasion/shooter'
37.    size_hint: None,None
38.    size: 40,40
39.    pos: self.parent.width/2, 0
```

```
40. <Boom@Image>:
41.    source: 'atlas://img/invasion/boom'
42.    size_hint: None,None
43.    size: 26,30
44. <Shot@Ammo>:
45.    source: 'atlas://img/invasion/shot'
46.    size_hint: None,None
47.    size: 12,15
48. <Missile@Ammo>:
49.    source: 'atlas://img/invasion/missile'
50   size_hint: None,None
51.    size: 12,27
```

Missile and Shot inherit from the same class called Ammo, which also inherits from Image. There is also the Boom class that will create the effect of explosion when any Ammo is triggered. Apart from the image (a star in the atlas), Boom will be associated with a sound that we are going to add in the next section.

Boom – simple sound effects

Adding sound effects in Kivy is very simple. Here is the code for boom.py, which is going to produce a sound every time a Shot or Missile is fired:

```
52. # File name: boom.py
53. from kivy.uix.image import Image
54. from kivy.core.audio import SoundLoader
55.
56. class Boom(Image):
57.    sound = SoundLoader.load(boom.wav')
58.    def boom(self, **kwargs):
59.      self.__class__.sound.play()
60.      super(Boom, self).__init__(**kwargs)
```

Reproducing a sound involves the use of two classes, Sound and SoundLoader (line 54). SoundLoader loads an audio file (.wav) and returns a Sound instance (line 57) that we keep in the sound reference (a static attribute of the Boom class). We play sound every time a new Boom instance is created.

Ammo – simple animation

This section explains how to animate Shots and Missiles, which show very similar behavior. They move from their original point to a destiny, constantly checking whether a target has been hit. The following is the code for the ammo.py class:

```
61. # File name: ammo.py
62. from kivy.animation import Animation
63. from kivy.uix.image import Image
64. from boom import Boom
65.
66. class Ammo(Image):
67.   def shoot(self, tx, ty, target):
68.     self.target = target
69.     self.animation = Animation(x=tx, top=ty)
70.     self.animation.bind(on_start = self.on_start)
71.     self.animation.bind(on_progress = self.on_progress)
72.     self.animation.bind(on_complete = self.on_stop)
73.     self.animation.start(self)
74.
75.   def on_start(self, instance, value):
76.     self.boom = Boom()
77.     self.boom.center=self.center
78.     self.parent.add_widget(self.boom)
79.
80.   def on_progress(self, instance, value, progression):
81.     if progression >= .1:
82.       self.parent.remove_widget(self.boom)
83.     if self.target.collide_ammo(self):
84.       self.animation.stop(self)
85.
86.   def on_stop(self, instance,value):
87.     self.parent.remove_widget(self)
88.
89. class Shot(Ammo):
90.   pass
91. class Missile(Ammo):
92.   pass
```

For the Ammo animation, we require a simple Animation (line 69). We send x and top as parameters. The parameters can be any property of the Widget to which we apply the animation. In this case, the x and top properties belong to Ammo itself. That is enough to set Animation of the Ammo from its original position to tx, ty.

> By default, the execution period of Animation is one second.

We need `Ammo` to do a few more things in its trajectory.

> 📝 The `Animation` class includes three events, which are triggered when the animations starts (`on_start`), during its progress (`on_progress`), and when it stops (`on_stop`).

We bind those events (line 70 to 72) to our own methods. The `on_start` method (line 75) displays (line 41) a `Boom` when the animation starts. The `on_progress` (line 80 to 84) method removes the `Boom` after 10 percent of `progression` (line 81 and 82). Also it is constantly checking the `target` (line 83). When the target is hit, the animation is stopped (line 84). Once the animation ends (or is stopped), the `Ammo` is removed from the parent (line 87).

Lines 89 to 92 define two classes, `Shoot` and `Missile`. `Shot` and `Missile` inherit from `Ammo` and their only difference is in the `images.kv`.

Invader – transitions for animations

The previous section uses the default `Animation` transition. This is a `Linear` transition, which means that the `Widget` instance moves from one point to another in a straight line. Invaders trajectories can be more interesting. For example, there could be accelerations, or changes of direction, as the following screenshot shows with the yellow line:
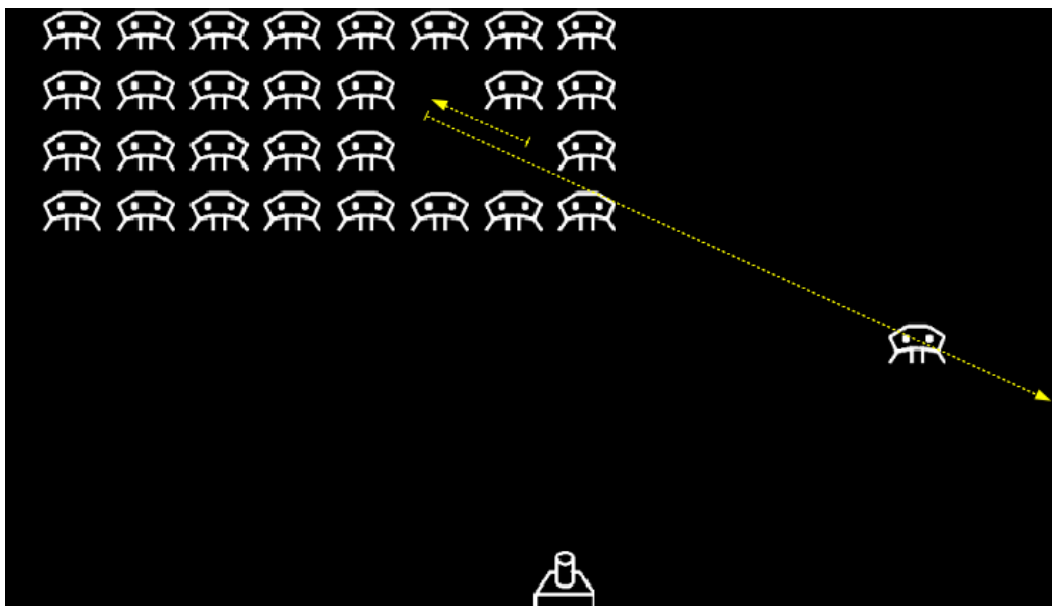
The following is the code of the `invader.py`:

```
93.  # File name: invader.py
94.  from kivy.core.window import Window
95.  from kivy.uix.image import Image
96.  from kivy.animation import Animation
97.  from random import choice, randint
98.  from ammo import Missile
99.
100. class Invader(Image):
101.    pre_fix = ['in_','out_','in_out_']
102.    functions = ['back','bounce','circ','cubic',
103.       'elastic','expo','quad','quart','quint','sine']
104.    formation = True
105.
106.    def solo_attack(self):
107.       if self.formation:
108.          self.parent.unbind_invader()
109.          animation = self.trajectory()
110.          animation.bind(on_complete = self.to_dock)
111.          animation.start(self)
112.
113.    def trajectory(self):
114.       fleet = self.parent.parent
115.       area = fleet.parent
116.       x = choice((-self.width,area.width+self.width))
117.       y = randint(round(area.y), round(fleet.y))
118.       t = choice(self.pre_fix) + choice(self.functions)
119.       return Animation(x=x, y=y,d=randint(2,7),t=t)
120.
121.    def to_dock(self, instance, value):
122.       self.y = Window.height
123.       self.center_x = Window.width/2
124.       animation = Animation(pos=self.parent.pos, d=2)
125.       animation.bind(on_complete =
    self.parent.bind_invader)
126.       animation.start(self)
127.
128.    def drop_missile(self):
129.       missile = Missile()
130.       missile.center = (self.center_x, self.y)
131.       fleet = self.parent.parent
132.       fleet.invasion.add_widget(missile)
133.       missile.shoot(self.center_x,0,fleet.shooter)
```

Sometimes, an Invader can break formation from the Fleet and proceed into a `solo_attack` (line 106 to 111) method. The Invader's `Animation` is created (lines 113 and 119) by randomizing the final point of the Invaders trajectory (a point outside of the screen next to the Enemy Area) on lines 116 and 117.

We also randomize the transition (line 118) and the duration (line 119).

> Kivy currently includes 31 transitions. They are represented by a string like `'in_out_cubic'`, where in_out is a prefix that describes the way in which the function (`cubic`) is used. There are three possible prefixes (`in`, `out`, and `in_out`), and 10 functions (line 102), such as `cubic`, `exponential`, `sin`, `quadratic`. Please visit the Kivy API for a description of all of them (`http://kivy.org/docs/api-kivy.animation.html`).

Line 118 selects one of the transitions randomly. The transition is applied to the progress, and therefore to `x` and `y` at the same time, which produces an interesting effect on the trajectories.

When the `Animation` class ends its trajectory (line 110), the `to_dock` method (lines 121 to 126) brings the Invader back to its original position starting from the top-center part of `Window`. We use the `Window` class to get `height` and `width`. Sometimes it is clearer than to traverse the chain of parents. When the Invader reaches the Dock, it is bound back to it (line 125). The last method (`drop_missile` on lines 128 to133) shoots one Missile that follows a vertical line starting from the Invader's bottom center position (line 130) to the bottom of the screen (line 133).

Dock – automatic binding in the Kivy language

You might realize from previous chapters that Kivy language does more than simply transforming its rules to Python instructions. For instance, you might see that when it creates properties, it also binds them.

> When we do something common like `pos: self.parent.pos` inside a layout, then the property of the parent is bound to its child. The child always moves to the parent position when the parent moves.

This is usually desirable but not all the time. Think about the `solo_attack` of the Invader. We need it to break formation and follow a free trajectory on the screen. While this happens, the whole formation of Invaders continues moving from right to left and vice versa. This means that the Invader will receive two orders at the same time. One from the moving parent and another from the trajectory's `Animation`.

That means that we need a placeholder (the Dock) for each Invader. This will secure the space for the Invader when it comes back from executing a solo attack, otherwise the `GridLayout` will automatically reconfigure the formation, reallocating the fleet to fill the empty space. Second, the Invader needs to free itself from the parent (the Dock) so it can float to any location on the screen. The following code (`dock.py`) explains how to manually bind (lines 145 to 147) and unbind (lines 149 to 151) the Invader:

```
134. # File name: dock.py
135. from kivy.uix.widget import Widget
136. from invader import Invader
137.
138. class Dock(Widget):
139.   def __init__(self, **kwargs):
140.     super(Dock, self).__init__(**kwargs)
141.     self.invader = Invader()
142.     self.add_widget(self.invader)
143.     self.bind_invader()
144.
145.   def bind_invader(self, instance=None, value=None):
146.     self.invader.formation = True
147.     self.bind(pos = self.on_pos)
148.
149.   def unbind_invader(self):
150.     self.invader.formation = False
151.     self.unbind(pos = self.on_pos)
152.
153.   def on_pos(self, instance, value):
154.     self.invader.pos = self.pos
```

We use knowledge from *Chapter 3, Widget Events – Binding Actions,* for this code, but the important part is the strategy that we apply. There will be situations in which we want to avoid using the Kivy language, because it is preferable to have complete control. That doesn't mean that it is impossible to solve this using Kivy language. For example, one common approach is to switch the Invader's parent (Dock) to, let's say, the root `Widget` of the application that unbinds the position of the Invader from its current parent. It doesn't really matter which approach we follow, as long as we are aware that we are taking complete control of it.

Fleet – infinite concatenation of animations

In the last section we studied how we can concatenate two animations. What happens if we want to have an object that is constantly moving from right to left and vice versa? This is the case of the `Fleet`: we want it constantly moving from left to right, like the yellow arrows of the following screenshot cut show:

> We can concatenate two animations with the `on_complete` event.

The following code fragment 1 (of 2) of `fleet.py` shows how to concatenate these events:

```
155. # File name: fleet.py (Fragment 1)
156. from kivy.uix.gridlayout import GridLayout
157. from kivy.properties import ListProperty
158. from kivy.animation import Animation
159. from kivy.clock import Clock
160. from kivy.core.window import Window
161. from random import randint, random
162. from dock import Dock
163.
164. class Fleet(GridLayout):
165.   survivors = ListProperty(())
166.
167.   def __init__(self,  **kwargs):
168.     super(Fleet, self).__init__(**kwargs)
169.     for x in range(0, 32):
170.       dock = Dock()
```

```
171.         self.add_widget(dock)
172.         self.survivors.append(dock)
173.     self.center_x= Window.width/4
174.
175.   def start_attack(self, instance, value):
176.     self.invasion.remove_widget(value)
177.     self.go_left(instance, value)
178.     self.schedule_events()
179.
180.   def go_left(self, instance, value):
181.     animation = Animation(x = 0)
182.     animation.bind(on_complete = self.go_right)
183.     animation.start(self)
184.
185.   def go_right(self, instance, value):
186.     animation = Animation(right=self.parent.width)
187.    animation.bind(on_complete = self.go_left)
188.     animation.start(self)
```

The `go_left` method (lines 180 to 183) binds the `on_complete` (line 182) event to the `go_right` method (lines 185 to 188). Similarly, the `go_right` method binds the `go_left` method to the new `Animation` (line 187). With this strategy, we create an infinite loop of two animations.

The `fleet.py` class also overloads the constructor to add 32 Invaders (lines 169 to 173) to the children of `Fleet`. These Invaders are added to the `survivors` `ListProperty` that we use to keep track of the Invaders that haven't been shot down. The `start_attack` method starts the `Fleet` animation calling on `go_left` and the `schedule_events` method. The latter makes use of the `Clock`, which is explained in the next section.

Scheduling events with the Clock

You might have noticed that `Animation` has a duration parameter that establishes the time in which an animation should take place. A different time-related problem is the scheduling of a particular task to start at a certain time, or during an interval of n seconds. In these cases, we use the `Clock` class. Let's analyze the following code, fragment 2 of `fleet.py`:

```
189. # File name: fleet.py (Fragment 2)
190.   def schedule_events(self):
191.     Clock.schedule_interval(self.solo_attack, 2)
192.     Clock.schedule_once(self.shoot,random())
193.
```

```
194.    def solo_attack(self, dt):
195.      if len(self.survivors):
196.        rint = randint(0, len(self.survivors) - 1)
197.        child = self.survivors[rint]
198.        child.invader.solo_attack()
199.
200.    def shoot(self, dt):
201.      if len(self.survivors):
202.        rint = randint(0,len(self.survivors) - 1)
203.        child = self.survivors[rint]
204.        child.invader.drop_missile()
205.        Clock.schedule_once(self.shoot,random())
206.
207.    def collide_ammo(self, ammo):
208.      for child in self.survivors:
209.        if child.invader.collide_widget(ammo):
210.          child.canvas.clear()
211.          self.survivors.remove(child)
212.          return True
213.      return False
214.
215.    def on_survivors(self, instance, value):
216.      if len(self.survivors) == 0:
217.        Clock.unschedule(self.solo_attack)
218.        Clock.unschedule(self.shoot)
219.        self.invasion.end_game("You Win!")
```

The `schedule_events` method (lines 190 to 192) schedules actions for a particular time. Line 191 schedules the `solo_attack` method every two seconds. Line 192 schedules the `shoot` just once at random (between 0 and 1) seconds.
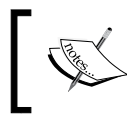
> The `schedule_interval` method schedules actions periodically, whereas the `schedule_once` method schedules an action just once.

The `solo_attack` method randomly selects one of the survivors to perform the solo attack that we studied for the Invaders (lines 195 to 198). The `shoot` method randomly selects one survivor to fire a Missile at the Shooter (lines 201 to 204). After that, the `shoot` method schedules another `shoot` (line 205).

We've used `collide_ammo` from the start to verify whether `Ammo` hits any of the Invaders (line 83 of `ammo.py`), in which case it is hidden and removed from the survivors list. The `on_survivors` is an event, triggered every time we modify the survivors `ListProperty`. When there are no survivors left, we unschedule the events (lines 217 and 218) and end the game by displaying a `You Win` message.

Shooter – multitouch control

Kivy supports multitouch interactions. This feature has been there since the very beginning but we didn't pay attention to it until now. We did use the multitouch Kivy features with `Scatter` in the previous chapter, however, we didn't clarify that the whole screen and GUI components are already multitouch, and Kivy handles the events accordingly.

> Kivy handles multitouch actions internally. This means that all the Kivy widgets and components support multitouch behavior; we don't have to worry about them. Kivy solves all the possible conflicts of ambiguous situations that are common in multitouch control, for example, touch two buttons at the same time.

However, it is up to us to control particular implementations. Multitouch programming introduces logic problems that we need to solve as programmers rather than a new set of tools that you have to learn. Nevertheless, Kivy provides the data related to each particular touch so we can work on the logic. The main problem is that we need to constantly distinguish one touch from another, and then take the respective actions.

With **Invaders Revenge,** we need to distinguish between two actions that are triggered by the same type of touch. The first action is the Shooter's horizontal movement in order to avoid the invaders' Missiles. The second is to fire at the Invaders. The following screenshot illustrates these two actions with the wide yellow arrows (sliding touch) and the dotted yellow arrow (shot action).
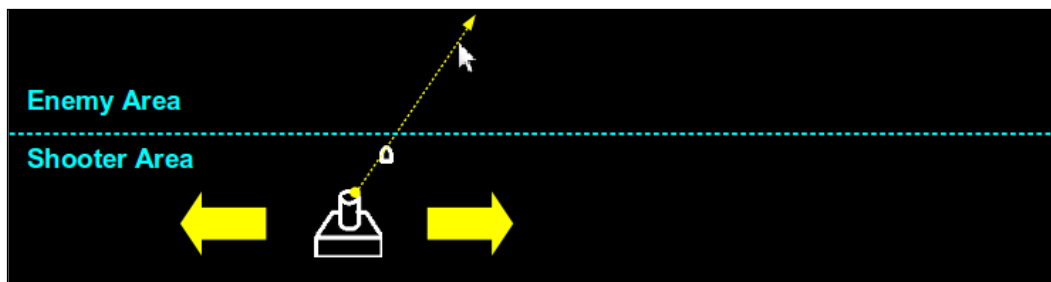
The following code (`shooter.py`) controls these two actions by using the two areas indicated in cyan color:

```
220.        # File name: shooter.py
221. from kivy.clock import Clock
222. from kivy.uix.image import Image
223. from ammo import Shot
224.
225. class Shooter(Image):
226.   reloaded = True
227.
228.   def on_touch_down(self, touch):
229.     if self.parent.collide_point(*touch.pos):
230.       self.center_x = touch.x
231.       touch.ud['move'] = True
232.     elif self.enemy_area.collide_point(*touch.pos):
233.       self.shoot(touch.x,touch.y)
234.       touch.ud['shoot'] = True
235.
236.   def on_touch_move(self, touch):
237.     if self.parent.collide_point(*touch.pos):
238.       self.center_x = touch.x
239.     elif self.enemy_area.collide_point(*touch.pos):
240.       self.shoot(touch.x,touch.y)
241.
242.   def on_touch_up(self, touch):
243.     if 'shoot' in touch.ud and touch.ud['shoot']:
244.       self.reloaded = True
245.
246.   def shoot(self, fx, fy):
247.     if self.reloaded:
248.       self.reloaded = False
249.       Clock.schedule_once(self.reload_gun, .5)
250.       shot = Shot()
251.       shot.center = (self.center_x, self.top)
252.       self.invasion.add_widget(shot)
253.       (fx,fy) =
            self.project(self.center_x,self.top,fx,fy)
254.       shot.shoot(fx,fy,self.invasion.fleet)
255.
256.   def reload_gun(self, dt):
257.     self.reloaded = True
258.
```

```
259.    def collide_ammo(self, ammo):
260.      if self.collide_widget(ammo) and self.parent:
261.        self.parent.remove_widget(self)
262.        self.invasion.end_game("Game Over")
263.        return True
264.      return False
265.
266.    def project(self,ix,iy,fx,fy):
267.      (w,h) = self.invasion.size
268.      if ix == fx: return (ix, h)
269.      m = (fy-iy) / (fx-ix)
270.      b = iy - m*ix
271.      x = (h-b)/m
272.      if x < 0: return (0, b)
273.      elif x > w: return (w, m*w+b)
274.      return (x, h)
```

The `on_touch_down` (lines 228 to 234) and `on_touch_move` (lines 236 to 240) methods distinguish between two actions (moving and shooting) by using the Shooter Area (lines 229 and 237) and the Enemy Area (lines 232 and 239) widgets to collide the coordinates of the event.

> The touch coordinates are the most common strategy to identify specific touches. However, touches have many other attributes that could help to distinguish between them, for example, timing, a double (or triple) tap, or the input device.

The `on_touch_up` method follows a different approach. It uses the `ud` attribute of a touch to distinguish if the touch down that started the event was a movement or a shoot. We set the `touch.ud` (lines 231 and 234) previously on `on_touch_down`.

> Kivy keeps the touch event associated with the three basic touch events (down, move, and up), so the touch references we get for `on_touch_down`, `on_touch_move`, and `on_touch_up` are the same, and we can distinguish between touches.

We implement an interesting behavior for the `on_touch_move` method with the `shoot` method (lines 246 to 254). Instead of shooting as fast as possible, we delay the next `shoot` by `0.5` seconds because the gun needs to be reloaded (line 249) and it would be unfair towards the Invaders if we didn't. When we use the `on_touch_up` method, the gun is reloaded immediately so we can always shoot faster with a touch-down and touch-up sequence.

The `collide_ammo` method (lines 259 to 264) is almost equivalent to the `collide_ ammo` method of the `Fleet` (lines 207 to 213). The only difference is that there is just one Shooter instead of a set of Invaders. And if the Shooter is hit, then the game is over and the message `Game Over` is displayed. The `project` method (lines 266 to 273) extents (project) the touch coordinates to the border of the screen, so the Shot will continue its trajectory until the end of the screen and not stop exactly at the touch coordinate.

Invasion – moving the shooter with the keyboard

This section offers a second possibility of how to move the Shooter. If you don't have a multitouch device, you will need to use something else to control the position of the Shooter easily while you use the mouse to shoot. The following code presents fragment 1 (of 2) of `main.py`:

```
275. # File name: main.py (Fragment 1)
276. from kivy.app import App
277. from kivy.lang import Builder
278. from kivy.core.window import Window
279. from kivy.uix.floatlayout import FloatLayout
280. from kivy.uix.label import Label
281. from kivy.animation import Animation
282. from kivy.clock import Clock
283. from fleet import Fleet
284. from shooter import Shooter
285.
286. Builder.load_file('images.kv')
287.
288. class Invasion(FloatLayout):
289.
290.   def __init__(self, **kwargs):
291.     super(Invasion, self).__init__(**kwargs)
292.     self._keyboard = Window.request_keyboard(self.close,
        self)
293.     self._keyboard.bind(on_key_down=self.press)
294.     self.start_game()
295.
296.   def close(self):
297.     self._keyboard.unbind(on_key_down=self.press)
298.     self._keyboard = None
299.
```

```
300.    def press(self, keyboard, keycode, text, modifiers):
301.      if keycode[1] == 'left':
302.        self.shooter.center_x -= 30
303.      elif keycode[1] == 'right':
304.        self.shooter.center_x += 30
305.      return True
306.
307.    def start_game(self):
308.      label = Label(text='Ready!')
309.      animation = Animation (font_size = 72, d=2)
310.      animation.bind(on_complete=self.fleet.start_attack)
311.      self.add_widget(label)
312.      animation.start(label)
```

The code we just saw illustrates the keyboard event control. The `__init__` constructor (lines 290 to 294) will request `keyboard` (line 292) and `bind` (line 293) the `on_keyboard_down` method to the `press` method. One important parameter of the `Window._request_keyboard` method is the method that is called on when `keyboard` is closed (lines 296 to 298). There are many reasons why this can happen, including when another widget is requesting it. The `press` method (lines 300 to 305) is the one in charge of handling the keyboard input, the pressed key. The pressed key is kept in the `keycode` parameter and it is used in lines 301 and 303 to decide whether the Shooter should move left or right.
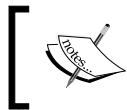
> The keyboard binding in the game is for testing purposes. If you want to try it on your mobile device, you should comment in lines 292 and 293 to deactivate the keyboard binding.

The line 294 calls the `start_game` method (lines 307 to 312). The method displays a `Label` with the text `Ready!` Notice that we applied `Animation` to `font_size` in line 309. So far, we have been using the animations to move widgets around with `x`, `y`, or `pos` properties. However, we said animations work with any property (that supports arithmetic operators). For example, we could even use them to animate the rotation or scaling of `Scatter`.

Combining animations with '+' and '&'

You have already learned that you can add several properties to the same animation so that they are modified together (line 69 of `ammo.py`).
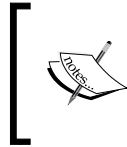
> We can combine animations by using the '+' and '&' operators. The '+' operator is used to create sequenced animations (one after another). The '&' operator lets us execute two animations at the same time.

The '+' operator is similar to what we do when we bind the `Animation on_complete` event to a method that creates another `Animation` in the Invader (line 110 of `invader.py`). The difference is that when we use the '+' operator, there is no chance to reset the `Widget` properties. In the Invader case, we relocated the Invader to the top-center (lines 122 and 123) of the screen before going back to the Dock.

The '&' operator is similar to sending two properties as parameters; the difference here is that they share neither the same duration, nor the same transition. For example, we could have used one transition for the x property and another for the y property of the Invader, instead of one for both (line 119 of `invader.py`), by joining two individual animations per property.

The following code is fragment 2 of `main.py`, and illustrates the use of these two operators:

```
313. # File name: main.py (Fragment 2)
314.   def end_game(self, message):
315.     label = Label(markup=True, size_hint = (.2, .1),
316.         pos=(0,self.parent.height/2), text = message)
317.     self.add_widget(label)
318.     self.composed_animation().start(label)
319.
320.   def composed_animation(self):
321.     animation = Animation (center=self.parent.center)
322.     animation &= Animation (font_size = 72, d=3)
323.     animation += Animation(font_size = 24,y=0,d=2)
324.     return animation
325.
326. class InvasionApp(App):
327.   def build(self):
328.     return Invasion()
329.
330. if __name__=="__main__":
331.   InvasionApp().run()
```

The end_game method (line 314 to 318) displays a final message to indicate how the game ended (You Win on line 219 of fleet.py or Game Over on line 262 of shooter.py). This method uses the composed_animation method (lines 320 to 324) to create a composed Animation, in which we use all the possibilities to combine animations. Line 321 is a simple Animation that is joined (with the '&' operator) to execute at the same time with another simple Animation of a different duration (line 322). In line 323, an Animation containing two properties (font_size and y) is attached to the previous one with the '+' operator.

The resulting animation does the following: it takes one second to move the message from left to middle, while the font size increases in size. When it gets to the middle, the increase of the size continues for two more seconds. Once the font reaches its full size (72 points), the message moves to the bottom and keeps decreasing in size at the same time. The following is one last screenshot which shows how the invaders have finally taken their revenge:
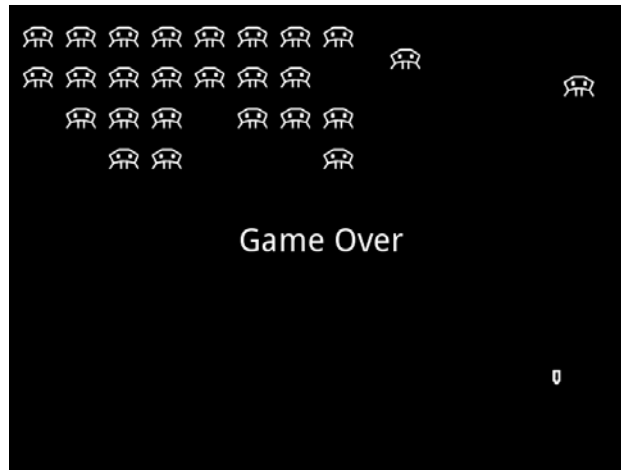
Summary

This chapter covered the whole construction process of an interactive and animated application. You learned how to integrate various Kivy components and you should now be able to comfortably build a 2D animated game.

Let's review all the new classes and components we used in this chapter:

- `Atlas`
- `Image: source` property
- `SoundLoader` and `Sound: load` and `play` methods respectively
- `Window: height` and `width` properties; `request_keyboard`, `remove_widget`, and `add_widget` method
- `Animation:` properties as parameters; `d` and `t` parameters; `start`, `stop`, and `bind` methods; `on_start`, `on_progress`, and `on_complete` events; and `'+'` and `'&'` operators
- `Touch: ud` attribute
- `Clock: schedule_interval` and `schedule_once` methods
- `Keyboard: bind` and `unbind` methods, `on_key_down` event

The information contained in this chapter delivers tools and strategies to start with the development of highly interactive applications. In combination with the previous chapters, and the provided insights into the use of properties, binding events, and further understanding of the Kivy language, you should be able to quickly start using all other components of the Kivy API (`http://kivy.org/docs/api-kivy.html`).

The beginning is at the end. Now it's your turn to start your own application.

Index

Symbols

A

B

C

Thank you for buying
Kivy: Interactive Applications in Python

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
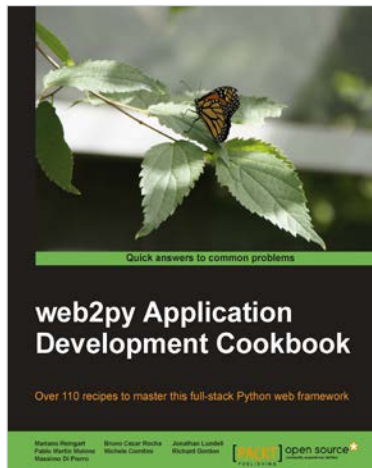
web2py Application Development Cookbook

ISBN: 978-1-849515-46-7 Paperback: 364 pages

Over 100 recipes to master this full-stack Python web framework

1. Take your web2py skills to the next level by dipping into delicious, usable recipes in this cookbook Learn advanced web2py usage from building advanced forms to creating PDF reports Written by developers of the web2py project with plenty of code examples for interesting and comprehensive learning

Sencha Architect App Development

ISBN: 978-1-782169-81-9 Paperback: 120 pages

Develop your own Ext JS and Sencha Touch application using Sencha Architect

1. Use Sencha Architect's features to improve productivity

2. Create your own application in Ext JS and Sencha Touch

3. Simulate, build, package and deploy your application using Sencha Command and Sencha Architect

Please check **www.PacktPub.com** for information on our titles
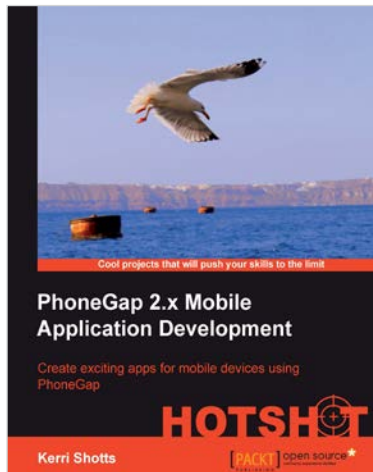
PhoneGap 2.x Mobile Application Development Hotshot

ISBN: 978-1-849519-40-3   Paperback: 388 pages

Create exciting apps for mobile using PhoneGap

1. Ten apps included to help you get started on your very own exciting mobile app

2. These apps include working with localization, social networks, geolocation, as well as the camera, audio, video, plugins, and more

3. Apps cover the spectrum from productivity apps, educational apps, all the way to entertainment and games

4. Explore design patterns common in apps designed for mobile devices

Developing Web Applications with Oracle ADF Essentials

ISBN: 978-1-782170-68-6   Paperback: 270 pages

Quickly build attractive, user-friendly web applications using Oracle's free ADF Essentials toolkit

1. Quickly build compete applications with business services, page flows, and data-bound pages without programming

2. Use Java to implement any business rule or application logic

3. Choose the right architecture for high productivity and maintainability

Please check **www.PacktPub.com** for information on our titles