KIVY - A Framework for **N**atural **U**ser **I**nterfaces

Nik Klever
Faculty of Computer Sciences
University of Applied Sciences Augsburg

Source of all Slides adopted from http://www.kivy.org

Kivy - Open Source Library

Kivy is an

- **Open Source Python library** for
- **rapid development** of applications that make use of
- **innovative user interfaces**, such as multi-touch apps.

Cross Platform

Kivy is running on

- **Linux,**
- **Windows,**
- **MacOSX,**
- **Android and**
- **IOS**

You can run the same code on all supported platforms.

It can use natively most input protocols and devices like

- WM_Touch, WM_Pen on **Windows**,
- Mac OS X Trackpad and Magic Mouse on **MacOSX**,
- mtdev, Linux Kernel HID, TUIO on **Linux**.

A multi-touch mouse simulator is included.

Business Friendly

Kivy is 100% **free to use**, under LGPL 3 licence. The toolkit is professionally developed, backed and used. You can use it in a product and sell your product.

The **framework is stable** and has a **documented API**, plus a programming guide to help for in the first step.

GPU accelerated

The graphics engine is built over **OpenGL ES 2,** using modern and fast way of doing graphics.

The toolkit is coming with more than 20 widgets designed to be extensible. Many parts are written in C using **Cython**, tested with regression tests.

Philosophy - 1

Fresh

Kivy is made for today and tomorrow. **Novel input methods** such as Multi-Touch have become increasingly important. Kivy is created from scratch, specifically for this kind of interaction.

Fast

Kivy is fast. This applies to both: **application development** and **application execution speeds**. Time-critical functionality in Kivy is implemented on the **C level** to leverage the power of existing compilers. Most importantly, we use the **GPU** wherever it makes sense in our context.

Flexible

Kivy is flexible. This means it can be run on a variety of different devices, including Android and iOS powered smartphones and tablets. We support all major operating systems (Windows, Linux, OS X). Kivy supports TUIO (Tangible User Interface Objects) and a number of other input sources.

Philosophy - 2

Focused

Kivy is focused. You can write a simple application with a **few lines of code**. Kivy programs are created using the Python programming language, which is incredibly versatile and powerful, yet easy to use. In addition, we created our own description language, the **Kivy Language**, for creating sophisticated user interfaces. This language allows you to set up, connect and arrange your application elements quickly.

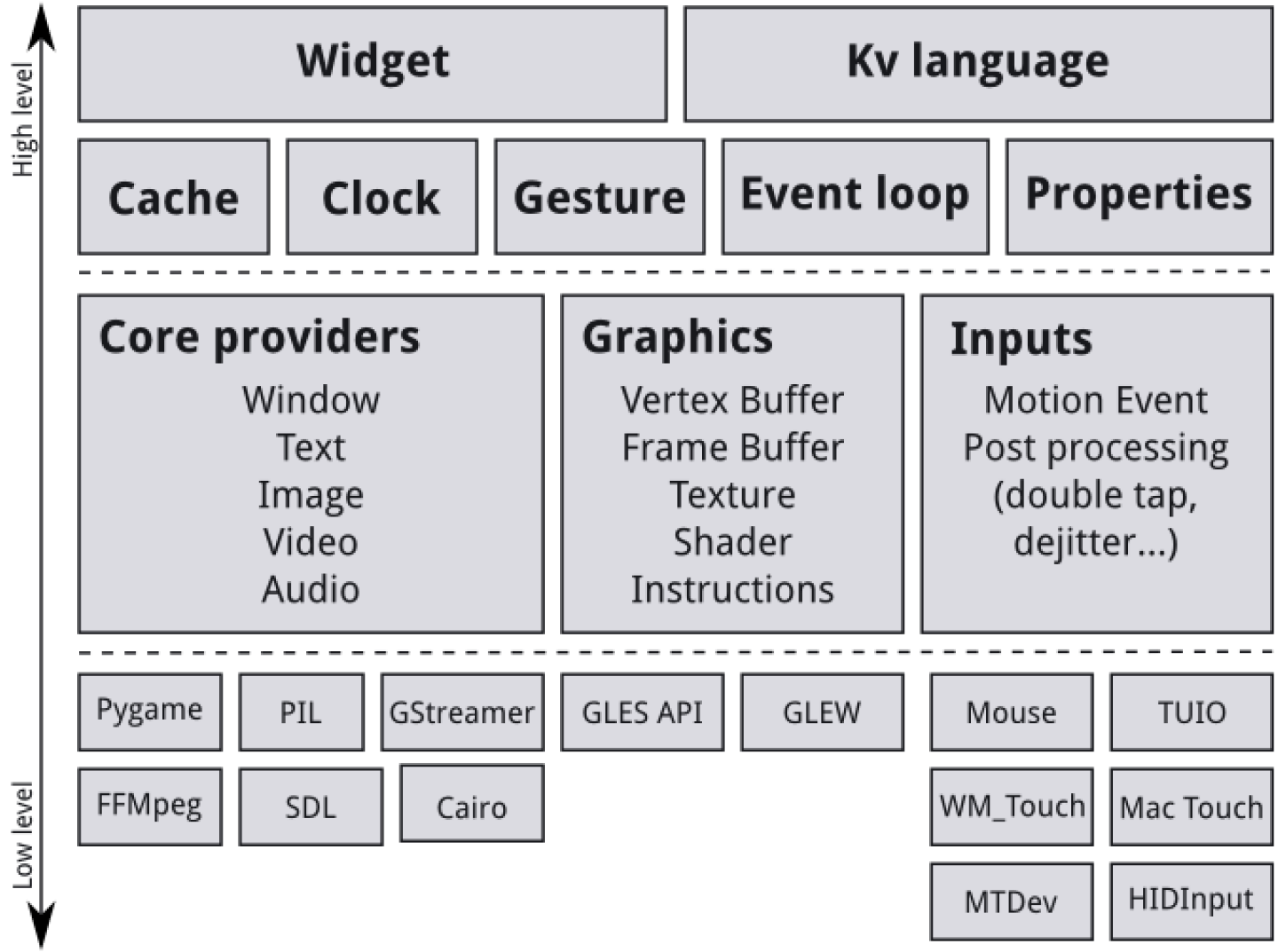
Funded

Kivy is actively **developed by professionals** in their field. Kivy is a community-influenced, professionally developed and commercially backed solution.

Free

Kivy is **free to use**. You don’t have to pay for it. You don’t even have to pay for it if you’re making money out of selling an application that uses Kivy.

Kivy Architecture



Core Providers and Input Providers

One idea that is key to understanding Kivy's internals is that of **modularity and abstraction**. Kivy tries to abstract basic tasks such as opening a window, displaying images and text, playing audio, getting images from a camera, spelling correction and so on. Kivy calls these core tasks. This makes the API both easy to use and easy to extend. A piece of code that uses one of these different native APIs to talk to the operating system on one side and to Kivy on the other (acting as an intermediate communication layer) is what Kivy calls a **core provider**.

The advantage of using specialized core providers for each platform is that Kivy can fully leverage the functionality exposed by the operating system and act as efficiently as possible and also reduces the size of the Kivy distribution. This also makes it easier to port Kivy to other platforms.

Kivy follows the same concept with input handling. An input provider is a piece of code that adds support for a specific input device, such as Apple's trackpads, TUIO or a mouse emulator. If you need to add support for a new input device, you can simply provide a new class that reads your input data from your device and transforms them into Kivy basic events.

Graphics

Kivy's **graphics API** is our abstraction of **OpenGL**. On the lowest level, Kivy issues **hardware-accelerated drawing commands** using OpenGL. Writing OpenGL code however can be a bit confusing, especially to newcomers. That's why we provide the graphics API that lets you draw things using simple metaphors that do not exist as such in OpenGL (e.g. Canvas, Rectangle, etc.).

All of our widgets themselves use this graphics API, which is **implemented on the C level for performance reasons.**

Another advantage of the graphics API is its ability to **automatically optimize the drawing commands** that your code issues. This is especially helpful if you're not an expert at tuning OpenGL. This makes your drawing code more efficient in many cases.

You can, of course, still use raw OpenGL commands if you prefer. The version we target is OpenGL 2.0 ES (GLES2) on all devices, so if you want to stay cross-platform compatible, we advise you to only use the GLES2 functions.

Core

The code in the core package provides commonly used features, such as:

- **Clock**
  You can use the clock to schedule timer events. Both one-shot timers and periodic timers are supported
- **Cache**
  If you need to cache something that you use often, you can use our class for that instead of writing your own.
- **Gesture Detection**
  We ship a simple gesture recognizer that you can use to detect various kinds of strokes, such as circles or rectangles. You can train it to detect your own strokes.
- **Kivy Language**
  The kivy language is used to easily and efficiently describe user interfaces.
- **Properties**
  These are not the normal properties that you may know from python. They are our own property classes that link your widget code with the user interface description.

UIX (Widgets & Layouts)

The UIX module contains commonly used widgets and layouts that you can reuse to quickly create a user interface.

- **Widgets**
  Widgets are user interface elements that you add to your program to provide some kind of functionality. They may or may not be visible. Examples would be a file browser, buttons, sliders, lists and so on. Widgets receive MotionEvents.
- **Layouts**
  You use layouts to arrange widgets. It is of course possible to calculate your widgets' positions yourself, but often it is more convenient to use one of our ready made layouts. Examples would be Grid Layouts or Box Layouts. You can also nest layouts.

Input Events (Touches)

Kivy **abstracts different input types** and sources such as touches, mice, TUIO or similar. What all of these input types have in common is that you can associate a 2D onscreen-position with any individual input event.

All of these input types are represented by instances of the Touch() class. A touch instance, or object, can be in one of three states. When a touch enters one of these states, your program is informed that the event occurred. The three states a touch can be in are:

- **Down**
  A touch is down only once, at the very moment where it first appears.
- **Move**
  A touch can be in this state for a potentially unlimited time. A touch does not have to be in this state during its lifetime. A ‘Move’ happens whenever the 2D position of a touch changes.
- **Up**
  A touch goes up at most once, or never. In practice you will almost always receive an up event because nobody is going to hold a finger on the screen for all eternity, but it is not guaranteed. If you know the input sources your users will be using, you will know whether or not you can rely on this state being entered.

Application

Creating a kivy application is as simple as:

- sub-classing the **App** class
- implementing its **build()** method so it returns a Widget instance (the root of your widget tree)
- instantiating this class, and calling its **run()** method.

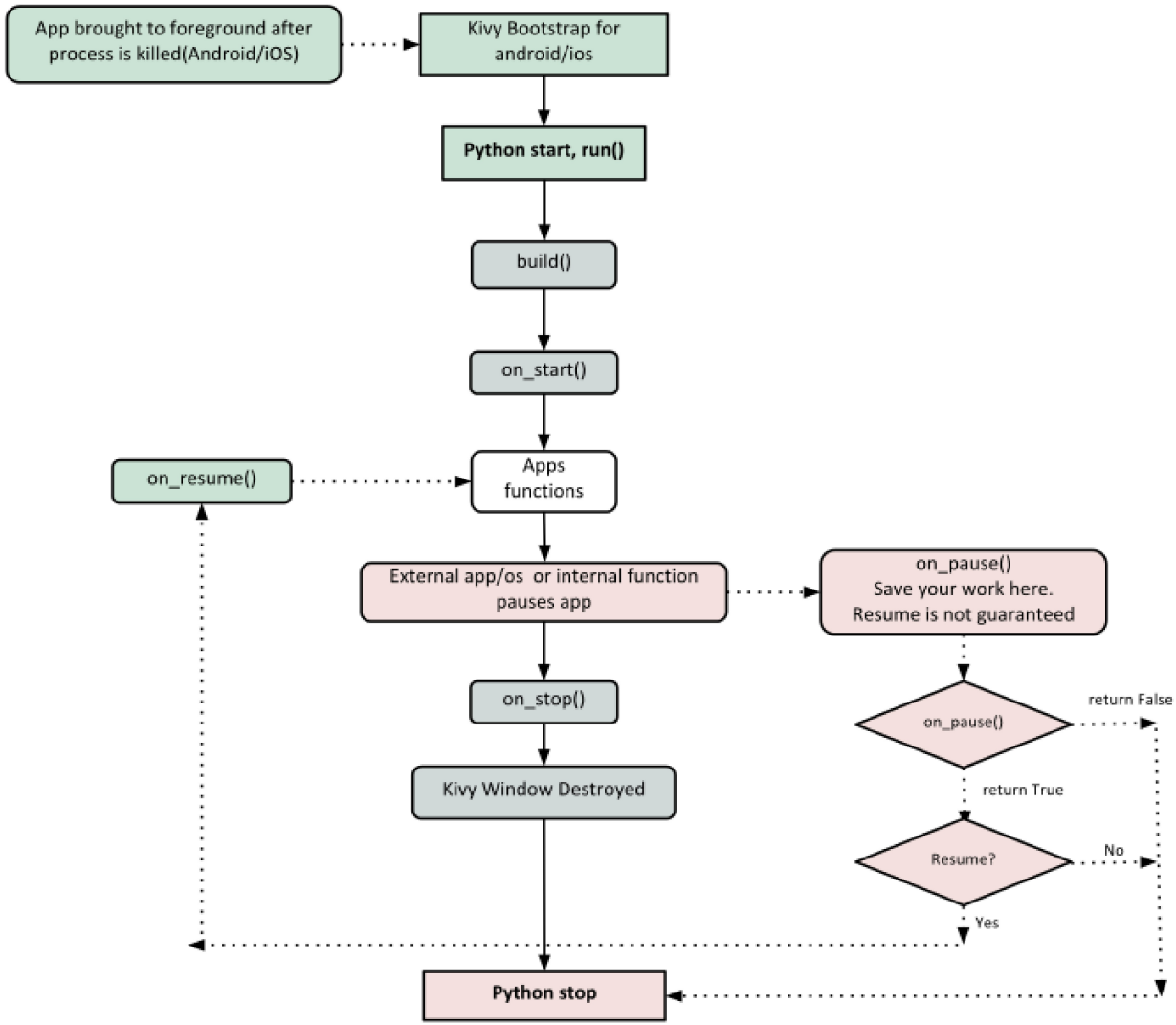
```
from kivy.app import App
from kivy.uix.label import Label

class MyApp(App):

    def build(self):
        root = Label(text="Hello World")
        return root

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

Application Life Cycle



Customizing the Application

Customizing this kivy application to introduce Username/Password Widgets:

- importing **GridLayout**, **Label** and **TextInput** classes

```
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.textinput import TextInput

class LoginScreen(GridLayout):
    def __init__(self, **kwargs):
        super(LoginScreen, self).__init__(**kwargs)
        self.rows = 2
        self.cols = 2
        self.padding = 10
        self.spacing = 10
        self.add_widget(Label(text='User Name:'))
        self.username = TextInput(multiline=False)
        self.add_widget(self.username)
        self.add_widget(Label(text='Password:'))
        self.password = TextInput(password=True, multiline=False)
        self.add_widget(self.password)

class MyApp(App):
    def build(self):
        return LoginScreen()

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

Basic Widget Concepts

Simple Paint App:

http://kivy.org/docs/tutorials/firstwidget.html

KV Language

Kivy provides a design language specifically geared towards easy and scalable GUI Design. The language makes it simple to separate the interface design from the application logic, adhering to the separation of concerns principle. For example the above loginscreen used with Kivy language:

```
every class in your app can be represented by a rule like the following in the kv file:
<LoginScreen>:
    f.username: username
    f.password: password
    this is how you add your widget/layout to the parent (note the indentation):
    GridLayout:
        this how you set each property of your widget/layout:
        rows: 2
        cols: 2
        padding: 10
        spacing: 10
        Label:
            text: 'User Name:'
        TextInput:
            id: username
            multiline: False
        Label:
            text: 'Password:'
        TextInput:
            id: password
            password: True
            multiline: False
```

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout

class LoginScreen(FloatLayout):
    pass

class LoginScreenApp(App):
    def build(self):
        return LoginScreen()

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


TextInput Events

To get access to the text, a user enters, we have use the **on_text_validate** event of the TextInput class, which is thrown when a user enters the „<CR>“ key, if the TextInput widget is not in multiline mode:

```
Label:
    text: 'User Name:'
TextInput:
    id: username
    multiline: False
    on_text_validate: root.do_action('username',username)

Label:
    text: 'Password:'
TextInput:
    id: password
    password: True
    multiline: False
    on_text_validate: root.do_action('password',password)
```

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout

class LoginScreen(FloatLayout):
    def do_action(self,key,value):
        print key,'=',value.text

class LoginScreenApp(App):
    def build(self):
        return LoginScreen()

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


Widgets and Event Dispatching

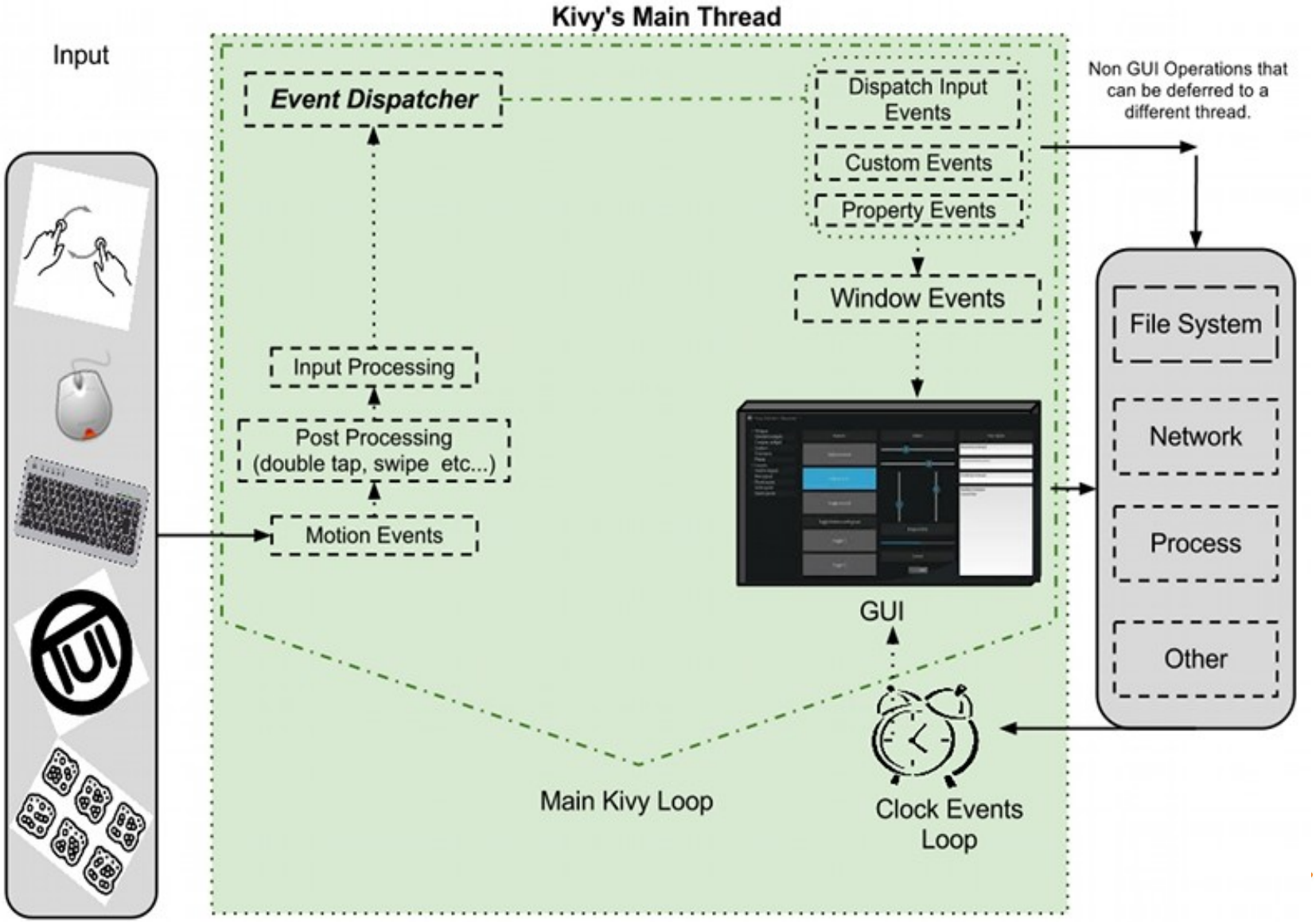
The term widget is often used in GUI programming contexts to describe some part of the program that the user interacts with. In Kivy, a **widget is an object that receives input events**. It does not necessarily have to have a visible representation on the screen. All widgets are **arranged in a widget tree**.

When new input data is available, Kivy sends out one event per touch. The root widget of the widget tree first receives the event. Depending on the state of the touch, the on_touch_down, on_touch_move or on_touch_up event is dispatched (with the touch as the argument) to the root widget, which results in the root widget's corresponding event handler being called.

Each widget (this includes the root widget) in the tree can choose to either digest or pass the event on. If an event handler returns True, it means that the event has been digested and handled properly. No further processing will happen with that event. Otherwise, the event handler passes the widget on to its own children by calling its superclass's implementation of the respective event handler.

Often times you will want to restrict the area on the screen that a widget watches for touches. You can use a widget's collide_point() method to achieve this. You simply pass it the touch's position and it returns True if the touch is within the 'watched area' or False otherwise. By default, this checks the rectangular region on the screen that's described by the widget's pos (for position; x & y) and size (width & height), but you can override this behaviour in your own class.

Events and Properties



Event Dispatcher

One of the most important base classes of the framework is the EventDispatcher class. This class allows you to register event types, and to dispatch them to interested parties (usually other event dispatchers). The **Widget, Animation** and **Clock** classes are examples of event dispatchers.

As outlined in the illustration above, Kivy has a main loop. It's important that you avoid breaking it. The main loop is responsible for reading from inputs, loading images asynchronously, drawing to frame, ...etc. Avoid long/infinite loops or sleeping. For example the following code does both:

```
while True:
    animate_something()
    time.sleep(.10)
```

When you run this, the program will never exit your loop, preventing Kivy from doing all of the other things that need doing. As a result, all you'll see is a black window which you won't be able to interact with. You need to "schedule" your animate_something() function call over time. You can do this in 2 ways: a repetitive call or one-time call.

Schedule

Scheduling a repetitive event

You can call a function or a method every X times per second using schedule_interval(). Here is an example of calling a function named my_callback 30 times per second:

```
def my_callback(dt):
    print 'My callback is called', dt
Clock.schedule_interval(my_callback, 1 / 30.)
```

Scheduling a one-time event

Using schedule_once(), you can call a function “later”, like in the next frame, or in X seconds:

```
def my_callback(dt):
    print 'My callback is called !'
Clock.schedule_once(my_callback, 1)
```

Trigger & Widget Events

Trigger Events:

```
trigger = Clock.create_trigger(my_callback)
# later
trigger()
```

Each time you call trigger, it will schedule a single call of your callback.
If it was already scheduled, it will not be rescheduled.

Widget Events:

A widget has 2 types of events:

- Property event: if your widget changes its position or size, an event is fired.
- Widget-defined event: an event will be fired for a Button when it’s pressed or released.

Properties

Properties are an awesome way to define events and bind to them. Essentially, they produce events such that when an attribute of your object changes, all properties that reference that attribute are automatically updated.

There are different kinds of properties to describe the type of data you want to handle.

- StringProperty
- NumericProperty
- BoundedNumericProperty
- ObjectProperty
- DictProperty
- ListProperty
- OptionProperty
- AliasProperty
- BooleanProperty
- ReferenceListProperty

Declaring Properties

Kivy introduces a new way of declaring properties within a class. Before:

```
class MyClass(object):
    def __init__(self):
        super(MyClass, self).__init__()
        self.numeric_var = 1
```

After, using Kivy's properties:

```
class MyClass(EventDispatcher):
    numeric_var = NumericProperty(1)
```

Theses properties implement the Observer pattern. They help you to:

- Easily **manipulate widgets** defined **in the Kv language**
- **Automatically observe any changes** and dispatch functions/code accordingly
- Check and validate values
- Optimize memory management

To use them, you have to declare them at class level. That is, directly in the class, not in any method of the class. A property is a class attribute that will automatically create instance attributes. Each property by default provides an on_<propertyname> event that is called whenever the property's state/value changes .

Tutorial Example **Pong**

Pong is an easy, small and very common example of the old Ping-Pong Computer Game – it is introduced as a tutorial for Kivy on the site

http://kivy.org/docs/tutorials/pong.html

What is needed ? Download Kivy as described on the Download Site

HowTo-Example on Android:

1. Download the Kivy Launcher App from the Google Play Store
2. Create a directory named **kivy** in the root-directory on the storage of your Android smartphone
3. Create a directory named **pong** in this directory
4. Copy the files **main.py** and **pong.kv** into this directory pong
5. Create a file name **android.txt** with the following content:

```
title=Pong Tutorial
author=KivyTeam
orientation=landscape
```

6. Run the application **Kivy Launcher** and select **Pong Tutorial**
7. Change pong.kv for your special needs (e.g. introduce a Color)

Questions ?

Kivy is well documented on it's Website http://kivy.org

Thanks to all contributors of Kivy, especially the core team

- Mathieu Virbel
- Thomas Hansen
- Gabriel Pettier