Human Computer Interaction

Events and event-driven programming

Prof. Andrew D. Bagdanov

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze
andrew.bagdanov AT unifi.it

October 17, 2017

Outline

Where we are, where we're going

The near future

Today: events and event-driven programming

- We will take a more detailed look at events and event models, using Kivy as our example framework.
- We will see how different event types can be used to communicate between widgets.
- And we will see how different binding mechanisms work.

Coming up:

- Tomorrow: Event lab. We will have a laboratory session dedicated to learning first hand how to manage asynchronicity.
- 20 October 2016: More events, and the KV design language.
- Next week: NO LESSONS (ICCV)

Homework/lab recap

Exercise 7.3: Natural conceptual models

Can you find me a better example than scissors? Try to find an object that perfectly and unambiguously communicates it purpose and how to use it.

GUI Lab: CounterButton

In this exercise you should implement a simple application that displays one or more buttons with the text "Clicks: N", where 'N' is the number of times that button has been clicked.

A running example

A running example from last week

- Recall the final laboratory exercise from last week.
- In it, we were asked to create a button that tracks the number of times it has been clicked.
- This was meant to be, essentially, an exercise in encapsulation of functionality in a derived widget class.
- We were asked to think carefully about where to put each element of the implementation:
  - Where should the callback be defined? At the class level? In which class?
  - Where should the counter value itself be stored? How should it be updated?
  - etc.

My first solution

- The original point of the exercise was to encapsulate the functionality of the click counter.
- As such, this CounterButton implementation does a pretty good job:

```
class CounterButton(Button):
    '''Simple button widget that counts number of times clicked.'''
    def __init__(self, initval=0, **kwargs):
        super(CounterButton, self).__init__(**kwargs)
        self._counter = initval
        self.text = 'Clicks: {}'.format(self._counter)
        self.bind(on_press=self.clicked)

    def clicked(self, instance):
        self._counter += 1
        self.text = 'Clicks: {}'.format(self._counter)
```

- As always, we should look at everything with a critical eye.
- What can be improved?

What works, what doesn't

- Considering the running example, there are a few things that work:
  - The encapsulation of functionality is fairly well-executed: no one needs to distinguish CounterButton from a standard `Button` in any way.
  - The callback `clicked()` doesn't get in the way, and is completely defined inside the extended `Button` class.
  - In other words, all functionality is encapsulated in the new implementation.
- There are some unsatisfactory elements of this example too:
  - The `Label` text update is redundant and should be automated in some way (or at least consolidated). Why?
  - Direct access to the `_counter` member variable is also not a great idea. Why?
  - As we will see, what is happening is we are mixing elements of the Model, the View, and the Controller.

The point of this example

- Development and design are iterative processes in which weaknesses are uncovered and incrementally eliminated.
- During this lecture we will see how some of these weaknesses can be addressed by more sophisticated events and event handling.
- This simple example (and an `Application` that uses it) of mine will serve as a running example that we will incrementally improve.
- This will serve as the basis for the laboratory tomorrow on events and event-driven programming.
- First, we take a step back and look at some basic concepts of events and event dispatching.

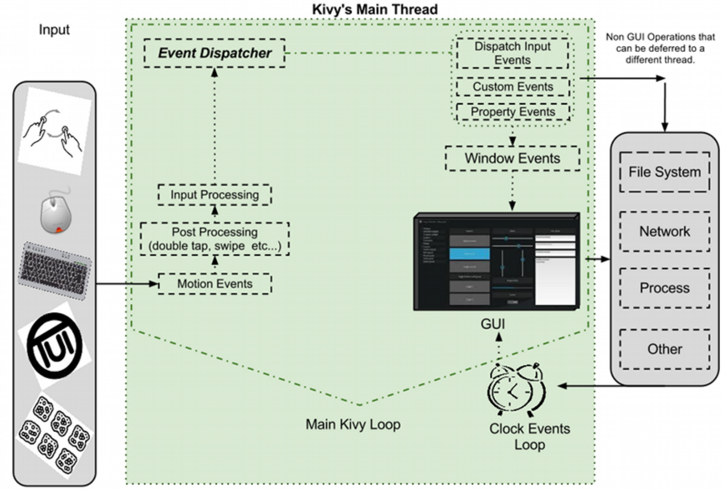
What is the 'main loop', really?

Conceptual model

- Recall that in most graphical user interface toolkits we invert control by calling the Application main loop.
- Conceptually, what goes on in this main loop?
- Well, we know that it calls the `build()` method of our application.
- And we know that it (somehow) handles the display and animation of all (visible) widgets in our built application.
- Another thing that the main loop must handle is event dispatching.
- A very approximate skeleton of what a main loop might look like:

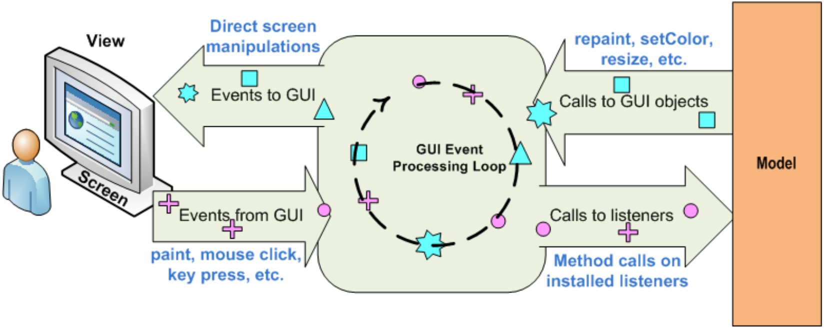
```
# Main application loop.
while True:
    animate_widgets(root_widget)     # Draws and animates visible widgets.
    user_input = get_user_input()    # Collects mouse, touch, key, etc, input

    # Dispatch user input events (which might generate other events).
    dispatch_events(root_widget, user_input)
    sleep(0.01)
```

- The figure below is the conceptual model of the actual Kivy event dispatcher.



Event dispatching (Java)

- For comparison, here is a diagram of the Java Swing event processing conceptual model:



The event pump 'gotcha'

- If there is one place in GUI-related code where we should be aware of efficiency it is in callback (or any event handling) code.
- This is because a GUI essentially ceases to function if the event pump isn't called.
- This terminology is used to evoke the image of pumping events through the system (like blood).
- Many GUI toolkits (I'm looking at you, GTK) require that you explicitly call the event pump.
- This is related to a different type of usage modality in which the application main loop cannot be called.
- OpenCV and Matlab are good examples of this.
- In any case, it is something that we should all be aware of.

Timer and other widget-less events

Timer and other widget-less events

- Let's first look at four classes of events common to GUI toolkits.
- Then, we will look at event binding mechanisms.
- We begin our tour of event types by looking at events that are derived from timers or other non-widget, non-user-input sources.
- These events provide a mechanism to schedule an event in the future.
- This scheduling can be done in several, flexible ways.

Scheduling something once

- The simplest timer event is one that is scheduled to run once in the future, N seconds from now.
- Kivy timer events are scheduled through the `kivy.clock.Clock` class.
- Timer event handlers take a single argument, which is the actual time elapsed from scheduling until the event was dispatched.

```python
# Callback to be called.
def onetime(dt):
    print('Onetime: I have been called!')

# Schedule event for 10 seconds in the future.
Clock.schedule_once(onetime, 10.0)

# Schedule event for *before* next frame (e.g. ASAP).
Clock.schedule_once(onetime, -1)

# Schedule event for *after* next frame.
Clock.schedule_once(onetime, 0)
```

Scheduling repetitive events

- More often, we want something to be called periodically (e.g. to poll some resource, to autosave, etc).
- Most GUI toolkits offer a timed event class that automatically re-schedules itself upon dispatch.
- In Kivy, use the `Clock.schedule_interval(callback, interval)` method to schedule `callback` every `interval` seconds.
- To remove a scheduled callback, use the `Clock.unschedule()` method.

```
def every_second(dt):
    print('I have been called at: {}'.format(dt))

Clock.schedule_interval(every_second, 1.0)
```

Trigger events

- Finally, sometimes you just want to trigger an event dispatch at the earliest possible convenience.
- This type of trigger event is useful for guaranteeing that certain events happen without having to render components aware of each other.
- In Kivy we use the `Clock.create_trigger()` method to create a callable trigger.
- When a trigger is called, it ensures that the associated callback is scheduled to run once and only once.

```
# Create a trigger that calls 'my_callback'
trigger = Clock.create_trigger(my_callback)

# Sometime later...
trigger()
```

Putting it all together

- A very simple application illustrating interval and one-shot timer events:

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

class MyApp(App):
    def build(self):
        return Label(text='Label')

def onetime(dt):
    Clock.unschedule(every_second)
    print('Onetime: I have been called at {}'.format(dt))

def every_second(dt):
    print('I have been called at: {}'.format(dt))

Clock.schedule_once(onetime, 10.0)
Clock.schedule_once(onetime, 10.0)
Clock.schedule_interval(every_second, 1.0)
Clock.schedule_interval(every_second, 1.0)
MyApp().run()
```

User input events

- The most common type of events pumping through the widget hierarchy are user input events.
- Kivy handles most types of input: mouse, touchscreen, accelerometer, gyroscope, etc.
- It handles native multitouch protocols on platforms: Tuio, WMTouch, MacMultitouchSupport, MT Protocol A/B and Android.
- The basic pipeline for input management is:
  1. Input providers: responsible for reading the input event from the operating system or the network.
  2. Motion event: the class of all input events, either a touch event (containing at least an X and a Y), or no-touch event like accelerometer data.
  3. Post processing: double- and triple-tap detection, etc.
  4. Dispatch: send motion events through the widget hierarchy.

A note on event binding

- In our previous examples we used the `bind()` method to bind a callback to specific events.
- Kivy provides a number of ways to associate callbacks to events, and binding is only one.
- A more standard way (one that Kivy also provides) is to override a default method in your derived class.
- These handlers (usually called `on_FOO()` for event type FOO) are automatically called:

```
class MyButton(Button):
    def on_press(self):
        print('Click!')

class MyLabel(Label):
    def on_touch_down(self, touch):
        if self.collide_point(*touch.pos):
            print('Touch down!')
```

Motion events

- We will concentrate on motion events, which are generated by touch (mouse movement or clicking) input by the user.
- There are three main event handlers for motion (touch) events:
  - `on_touch_down()`: which is signaled at the beginning of the touch and contains the absolute position of the touch.
  - `on_touch_move()`: which is signaled for every frame in which the touch persists and contains the absolute position (as well as dx, dy information.
  - `on_touch_up()`: which is signaled at the end of the touch and contains the absolute position where the touch ended.
- Motion input event handling is extremely complicated by the need to support many types of input devices: mouse, multitouch, Kinect, TUIO, etc.
- Each of these input devices provides a different profile of information.
- We can interrogate the `profile` attribute of the `touch` parameter to see what is present.

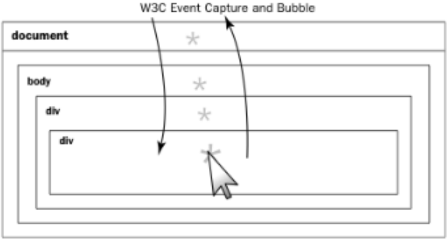
An illustrative example

- It is useful to look at a concrete example to understand how events move through the widget hierarchy:

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

class SensitiveLabel(Label):
    def on_touch_down(self, touch):
        print('{}: profile={}, button={}, pos={}'.format(self.text, touch.profile,
                                                          touch.button, touch.pos))
        return False

def my_touch(instance, touch):
    print('Me too!')
    return False

class MyApp(App):
    def build(self):
        box = BoxLayout(orientation='horizontal')
        box.bind(on_touch_down=my_touch)
        box.add_widget(SensitiveLabel(text='Left'))
        box.add_widget(SensitiveLabel(text='Right'))
        return box
```

Some observations

- The handler `on_touch_down()` returns a value (False in this case). What is this value?
- When we look at the coordinate outputs, we see that the coordinates are absolute values (i.e. they are not relative to the widget).
- Motion events are dispatched to all widgets, regardless of whether the event occurs within its physical boundaries on the screen.
- Note carefully the order in which events are dispatched.
- This process by which events are routed through the widget hierarchy is known as event bubbling.



Event bubbling

- Event bubbling (in Kivy) works by routing events top-down through the hierarchy of widgets:
  1. Sibling widgets are processed from most-recently-added first
  2. When processing a widget, events are bubbled-down from the widget to the leaves.
- This corresponds to in-order traversal of the widget hierarchy, where siblings are are stored in a stack (last-in, first-out).
- Normally, input events are geometrically filtered so that only events are propagated only to widgets whose physical extent collides with the event location.
- In Kivy, all motion events are propagated to all widgets, by default.
- Thus, we need mechanisms to:
  - Determine collision of events with widget extent.
  - Translate event coordinates to a local coordinate frame.
  - Capture events so they are not propagated everywhere.

Determining widget collision

- Kivy widgets provide a convenience method `collide_point(x, y)`:

```
In [1]: from kivy.uix.button import Button
# [... lots of deleted Kivy console spam ...]
In [2]: foo = Button()
# [... more deleted Kivy console spam ...]
In [3]: foo.collide_point?
Signature: foo.collide_point(x, y)
Docstring:
Check if a point (x, y) is inside the widget's axis aligned bounding
box.

:Parameters:
    `x`: numeric
        x position of the point (in window coordinates)
    `y`: numeric
        y position of the point (in window coordinates)

:Returns:
    A bool. True if the point is inside the bounding box, False
    otherwise.
```

Translating event coordinates

- Similarly, Kivy provides several convenience methods for converting from window to widget coordinate frames.
- Note that there are several translation functions, which return coordinates relative to different widget frames.

```
In [4]: foo.to_local?
Signature: foo.to_local(x, y, relative=False)
Docstring:
Transform parent coordinates to local coordinates. See
:mod:`~kivy.uix.relativelayout` for details on the coordinate systems.

:Parameters:
    `relative`: bool, defaults to False
        Change to True if you want to translate coordinates to
        relative widget coordinates.
```

Capturing events

- Finally, event handlers return a boolean value indicating whether the event has been processed (and thus does not need to be propagated further.
- This means that events can be intercepted so that they are propagated no further through the widget hierarchy.
- The choice between top-down bubbling and bottom-up bubbling is one of efficiency.
- Top-down ensures that (in general), the first and largest widgets are checked first.
- This can avoid propagating to very many small widgets which are irrelevant for a given event.

Widget-specific events

Widget-specific events

- We have already seen at least one type of widget-specific event: the `on_press` event from the `Button` class.
- Many widgets have specific events like this that are triggered when specific conditions (like the press of a button) occur.
- These events have default handlers that can be overridden in derived classes to implement custom behaviors.
- Or, as usual they can be bound using the `bind()` method to add a callback to a chain of handlers called when the event occurs.
- In most GUI toolkits, this is the primary way of defining event handlers.
- Many toolkits have a broad suite of widget events that can be captured: OnMouseMove, OnResize, OnExpose, etc.
- In Kivy, however, we will see that this method of handler definition is not used as much due to its flexible property event system.

Finding events

- Widget-specific events are so infrequent in Kivy, in fact, they can be difficult to find.
- The easiest way to locate widget events is by calling the `events()` method on an instance of the widget.
- Any of these can be bound or overridden to intercept the event.

```
In [1]: from kivy.uix.button import Button
In [2]: from kivy.uix.actionbar import ActionBar
In [3]: Button().events()
Out[3]:
['on_press',
 'on_touch_move',
 'on_touch_up',
 'on_release',
 'on_ref_press',
 'on_touch_down']
In [4]: ActionBar().events()
Out[4]: ['on_touch_up', 'on_touch_move', 'on_previous',
'on_touch_down']
```

Properties and property events

The way it's usually done

- As mentioned before, most GUI frameworks devote a lot of effort (and classes) to event handling.
- Specifically, to handling events related to the state of all widgets in the hierarchy.
- Every time the state changes, depending on the type of change (geometry, exposure, data) and specific type of event is generated.
- These events notify listeners that the state of the widget has changed in some way.
- This leads to a massive proliferation of event types and handlers in most GUI toolkits.

A general data-centric event system

- Kivy, however, takes a different approach to managing these types of events and their handlers.
- Kivy uses Properties as a 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.
- Kivy supports a rich variety of Properties: `StringProperty`, `NumericProperty`, `BoundedNumericProperty`, `ObjectProperty`, `DictProperty`, `ListProperty`, `OptionProperty`, `AliasProperty`, `BooleanProperty`, `ReferenceListProperty`.
- Note: these are not the same as object or class attributes or fields that you might be familiar with.

Properties, properties everywhere

- If we take a look at how most widget fields are defined at the class level, we will see that they are in face Properties.
- These pre-defined Properties can be bound to specific handler code that is called whenever their value changes:

```
In [1]: from kivy.uix.button import Button
In [2]: Button.pos
Out[2]: <ReferenceListProperty name=pos>
In [3]: Button.size
Out[3]: <ReferenceListProperty name=size>
In [4]: Button.width
Out[4]: <NumericProperty name=width>
In [5]: Button.height
Out[5]: <NumericProperty name=height>
```

Monitoring changes

- As with widget-specific events, there are two ways to bind handlers to events.
- We can use the `bind()` method on a Property instance, or we can implement a method in a derived class.
- The choice of which method to use can be a matter of necessity (because you have a widget instance and cannot extend the class).
- Or, it can be a matter of style.
- Note, however, that the semantics can be very different unless you are careful (see homework exercises for this lecture).

Property binding example

```
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.app import App

def on_resize(instance, newsize):
    print('New size (bind): {}'.format(newsize))

class ResizeButton(Button):
    def on_size(self, instance, newsize):
        print('New size (override): {}'.format(newsize))

    def on_pos(self, instance, newpos):
        print('New pos  (override): {}'.format(newpos))

class MyApp(App):
    def build(self):
        box = BoxLayout(orientation='horizontal')
        bl = Button(text='Button Left')
        bl.bind(size=on_resize)   # Explicit binding to size Property.
        box.add_widget(bl)
        box.add_widget(ResizeButton(text='Button Right'))
        return box

MyApp().run()
```

Rolling your own

- Of course, this system of properties and property change notifications is not limited to pre-defined widgets.
- You can include your own properties in custom widgets by declaring them at class level.
- Property classes ensure that all of the needed code to support binding is automatically generated in instances of the class.
- We will see a simple example of Property use right now as we return to our running example.

Returning to the running example

Returning to the running example

- We will now go back to our running example to see how we can use data-driven events to improve modularity and separation of concerns.
- Recall that we had a few problems with the original implementation.
- First, there was redundancy in Label update:

```
class CounterButton(Button):
    '''Simple button widget that counts number of times clicked.'''
    def __init__(self, initval=0, **kwargs):
        super(CounterButton, self).__init__(**kwargs)
        self._counter = initval
        self.text = 'Clicks: {}'.format(self._counter)  # Here.
        self.bind(on_press=self.clicked)

    def clicked(self, instance):
        self._counter += 1
        self.text = 'Clicks: {}'.format(self._counter)  # And here.
```

- We can immediately address the problem (well, one of them) with `CounterButton` using a `NumericProperty` for the counter and listening for changes.
- Here, we override the Property callback `on_counter` to implement our desired action.

```
class CounterButton(Button):
    '''Simple button widget that counts number of times clicked.'''
    counter = NumericProperty(-1)
    def __init__(self, initval=0, **kwargs):
        super(CounterButton, self).__init__(**kwargs)
        self.counter = initval
        self.bind(on_press=self.clicked)

    def on_counter(self, instance, pos):
        self.text = 'Clicks: {}'.format(self.counter)

    def clicked(self, instance):
        self.counter += 1
```

Reflections

General reflections

- In this lecture we saw how various types of events are generated, routed (bubbled), and bound to handler code.
- We saw how input events communicate the geometry of touch and motion events to handlers.
- We also saw how some widgets define custom, widget-specific events that can be caught.
- We saw how data-driven, or property events can be used as a general information passing mechanism in GUI applications.
- We saw how handler code can be bound in multiple ways to events: either by explicit binding through use of the `bind()` method, or through `overriding` in the definition of a derived class.

Kivy-specific reflections

- In this lecture we started with a running example in Kivy that illustrates how to incrementally refactor code.
- We saw how Kivy's Properties can be used to improve modularity and separation of concerns.
- Note that a large part of this flexibility on the part of Kivy is due to Python's dynamism and introspection capabilities.
- While this is extremely powerful and flexible in practice, care should be taken because it can lead to confusing object-oriented design (what's defined in the class definition, what's bound after-the-fact?).
- A good example of this is the event payload problem: how do we know what information is delivered to event handler code?
- Most GUI toolkits define a complex event hierarchy, each of which specifying the precise payload delivered by each event type.
- In Kivy, this information is only contained in the handler method signature.

Homework

Homework

Exercise 8.1: An extended running example

Before the laboratory tomorrow, spend some time studying the extended running example uploaded to the course Moodle. Make sure you can run it, and see how all event binding and interception is being done.