Visual Programming

College Of Computer Sciences and Mathematics

Dr. Firas Aswad

Dept. of Networks

Feb. 26, 2025

The Tkinter Geometry Manager

Three geometry managers in Tkinter let you specify the position of widgets inside a top-level or parent window.

The three geometry managers are:

1. **Pack**: It is simple to use for simpler layouts, but it may get very complex for slightly complex layouts.
2. **Grid**: This is the most commonly used geometry manager and provides a table-like layout of management features for easy layout management.
3. **Place**: This is the least popular, but it provides the best control for the absolute positioning of widgets.

The Place Geometry Manager

- The place geometry manager is the most rarely used geometry manager in Tkinter. It is not recommended, as it responds poorly to changes in window sizes, font sizes, and screen resolution.

- Nevertheless, it has its uses in that it lets you precisely position widgets within their parent frame by using the (x,y) coordinate system.

- The place manager can be accessed by using the place() method on any standard widget.

- The important options for place geometry include the following: Absolute positioning (specified in terms of x=N or y=N) Relative positioning (the key options include relx, rely, relwidth, and relheight).

- The other options that are commonly used with place include width and anchor(the default is NW).

Place When?

When should you use the place manager?

1. The place manager is useful in situations where you have to implement custom geometry managers.

2. If the widget placement is decided by the end user.

Note: While the pack and grid managers cannot be used together in the same frame, the place manager can be used with any geometry manager within the same container frame.

In Sum,

- The pack is suitable for a simple side-wise or top-down widget placement.
- The grid manager is best suited for the handling of complex layouts.
- The place manager is rarely used because, if you use it, you have to worry about the exact coordinates.

Events and Callbacks - adding life to programs

- The third component of GUI programming is about how to make widgets functional and responsive to events such as the pressing of buttons, the pressing of keys on a keyboard, and mouse clicks.

- To do so, we need to associate callbacks with specific events. Callbacks are normally associated with specific widget events using command-binding rules.

- Note: 1- component add widgets to a screen and 2- position.

1. Command Binding

The simplest way to add functionality to a button is called command binding, whereby a callback function is mentioned in the form of command = some_callback in the widget option.

Note that the command option is available only for a few selected widgets.

```
def my_callback ():
  # do something when the button is clicked

tk.Button(root, text="Click me", command=my_callback)
```

A callback is a function memory reference (my_callback in the preceding example) that is called by another function (which is Button in the preceding example), and that takes the first function as a parameter.

Passing Arguments to Callbacks

If a callback needs to take arguments, we can use the lambda function.

```
def my_callback (argument)
  #do something with the argument

tk.Button(root,text="Click", command=lambda: my_callback ('some argument'))
```

Note: The lambda function defines a single-line, nameless function on the fly.

Limitations of Command Option

The command option is only available with the Button widget and a few other widgets, such as the Checkbutton and Radiobutton Widget.

- The command button binds to the left-click and the spacebar. It does not bind to the Return key.
- This is counter-intuitive for many users. Also, you cannot change the binding of the command function easily.
- The command binding, though a very handy tool, is not flexible enough when it comes to deciding your own bindings.

2. Event Binding

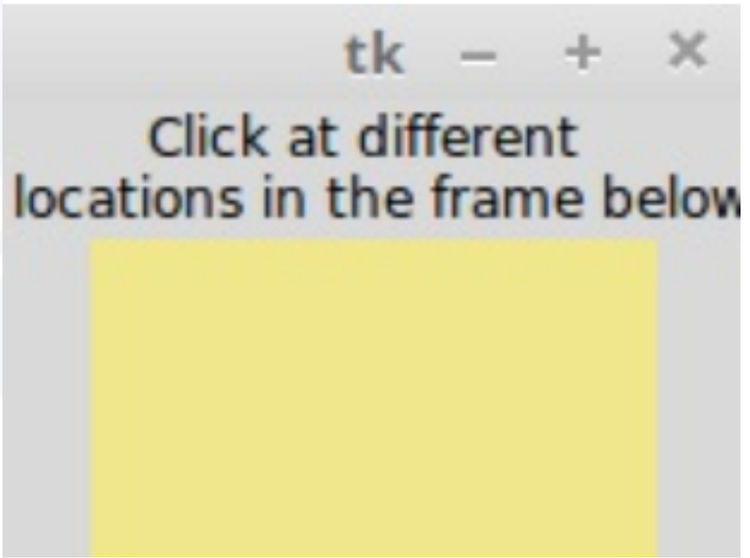
Tkinter provides an alternative event binding mechanism called bind() to deal with different events.

widget.bind(event, handler, add=None)

- When an event occurs in the widget, it calls not only the associated handler, which passes an instance of the event object as the argument, but also the details of the event, as in the example:

```
import tkinter as tk
root = tk.Tk()
tk.Label(root, text='Click at different \n locations in the frame below').pack()
def callback(event): ##(2)
   print(dir(event))##(3) Inspecting the instance event
   print("you clicked at", event.x, event.y )##(4)

frame = tk.Frame(root, bg='khaki', width=130, height=80)
frame.bind("<Button-1>", callback)##(1)
frame.pack()
root.mainloop()
```



Event Patterns

The event pattern	The associated event
• <Button-1>	Left-click of the mouse
• <KeyPress-B>	A keyboard press of the B key
• <Alt-Control-KeyPress-KP_Delete>	A keyboard press of Alt + Ctrl + Del

Homework! Use a P and left-click in the previous example

Event Pattern

An event pattern will comprise:

1. Event type
2. Event modifier
3. Event detail

Event Types

Some common event types include Button, ButtonRelease, KeyRelease, Keypress, FocusIn, FocusOut, Leave (when the mouse leaves the widget), and MouseWheel.

Below is the full list of event types:

Activate	Destroy	Map
ButtonPress, Button	Enter	MapRequest
ButtonRelease	Expose	Motion
Circulate	FocusIn	MouseWheel
CirculateRequest	FocusOut	Property
Colormap	Gravity	Reparent
Configure	KeyPress, Key	ResizeRequest
ConfigureRequest	KeyRelease	Unmap
Create	Leave	Visibility
Deactivate		

For more details visit: https://www.tcl-lang.org/man/tcl8.6/TkCmd/bind.htm#M7

Self-study page 68!

An event modifier

The event detail

Thank You!