The documentation of the library pygame_widgets

Jáchym Mierva

October 7, 2020

Contents

`pygame_widgets`

the root directory of the library

The root directory (better said the script `pygame_widgets\__init__.py`) includes the reference to the imported instance of `pygame`:

```
pygame_widgets.pygame
```

However, it is possible to import `pygame` separately. In that case it is not necessary to call `pygame.init()` as it was done during the `import pygame_widgets` command.

`new_loop(window=None) -> None`

announce a new iteration of the mainloop

This method contains various things that have to be done during every iteration of the mainloop. Firstly, the event `E_LOOP_STARTED` is published, another task is to handle the delayed functions called with `delayed_call()`.

In case the optional argument `window` is an instance of `pygame_widgets.Window` or its subclass, then the content of display is updated using `window.update_display()`.

Warning: It is strongly recommended to call `new_loop()` exactly once in every iteration of the mainloop, because the actions described above are important for proper function of `pygame_widgets`.

`delayed_call(func, delay=1, *args, **kwargs) -> None`

call a function with a delay

This function allows to call `func(*args, **kwargs)` after certain number of mainloop iterations. The number is specified by the argument `delay`. The number of mainloop cycles is determined by the number of calls to `new_loop()`; inside this function is also the call executed.

`set_mode_init() -> None`

block widgets to post events

This function was implemented after several fails caused by full event queue. When any attribute of any widget is changed by `widget.set()`, an event is posted. In case lots of widgets are initialized without processing events, the queue can be easily filled. Additionally, these events are often useless.

Calling this function will cause widgets not to post any events. This is recommended to do in the beginning of the program, right before initialization of widgets. To allow posting events again, use `set_mode_mainloop()`.

`set_mode_mainloop() -> None`

allow posting events

After calling this function widgets will be able to post events again (see `set_mode_init()`).

`get_mode() -> None`

returns `True` ***when widgets are allowed to post events, otherwise*** `False`***.***

Mode can be triggered using `set_mode_mainloop()` and `set_mode_init()`.

`pygame_widgets.widgets.widget._Master`

object, that contains attributes and methods common to all widgets

Warning: This object is not meant to be instanced or subclassed. Instancing will raise `TypeError`.

`on_screen(rect=None) -> bool`

returns `True` ***if there is image of*** `rect` ***on current window, otherwise*** `False`

In case no argument is provided, the return value of `self.get_abs_master_rect()` is used (this method is implemented in both `pygame_widgets.Window` and `pygame_widgets.widgets.widgets._Widgets`).

Warning: This method contains a bug, in specific cases returns `True` incorrectly.

`kwarg_list() -> list`

returns list of settable keyword attributes

The items of the return list are strings accepted as keys by the method `set()`.

`add_handler(event_type, func, args=None, kwargs=None, self_arg=True, event_arg=True, delay=0, index=None) -> None`

creates a new handler associated with given event type

`event_type` is the type of handled events, `func` is the handling function itself, `args` is an iterable with positional arguments passed to the `func`, `kwargs` is a mapping with keyword arguments. `self_arg` and `event_arg` are boolean arguments. They are used to specify, whether should the reference to the widget, whose handler was activated, and event, which is handled, be passed

to the handling function. In such case, these arguments would be inserted to the beginning of the `arg` list. `self` is always first. The parameter `delay` is non-negative integer specifying the number of iterations of the mainloop, that are going to be run through before calling the handler. This is possible to use to manage the order of activated handlers. The parameter `index` has similar purpose, it specifies the index of the new handler in the list of existing handlers (using the method `list.insert()`). In the default case, the handler is appended to the end of the list.

Still, the order control is temporary and it will be probably implemented differently in the future.

`remove_handler(event_type, func, args=None, kwargs=None, self_arg=True, event_arg=True) -> None`

Removes all handlers with specified attributes

All arguments must be identical to the arguments used to create the handler (see `_Master.add_handler()`.

`get_handlers(copy=True) -> dict`

returns dictionary with all handlers of the widget

The keys of the dictionary are event types and the corresponding values are lists of all handlers, which are (in order) called when handling an event. The handlers are instances of `pygame_wicgets.handler.Handler`.

In case `copy == False` the returned dictionary is a reference to the actual dictionary used by the widget. Therefore, user can change it and these changes reflect in the behaviour of the widget. The programmer has absolute control over the order of the handlers, on the other hand he might damage the basic functionality of the widget. This option should therefore be used only by experienced user, who knows, what he is doing.

In the default case the dictionary (including all lists and handlers) is only a copy. It is safe to change it, the changes don't influence the widget behaviour.

`add_grab(event_type, child, level=0) -> None`

send all events of the type `event_type` ***only to the widget*** `child`

The use of this method can be explained on an example. Suppose we are trying to implement an `Entry`. When an user clicks into it, the `entry` should be the only widget affected by the `KEYDOWN` events. This can be achieved by calling

```
entry.master.add_grab(pygame_widgets.constants.KEYDOWN, entry,
    -1)
```

The method is then called recursively - the immediate master of the `entry` also locks all `KEYDOWN` events only for itself etc. The integer `level` specifies

the depth of the recursion. In case it is a negative number, it goes all the way up to the absolute master widget.

Note: The widgets on the path to the `entry` do not handle the events.

`remove_grab(event_type, child, level=0) -> None`

cancel the effects of `add_grab()`

The parameters have the same meaning as in the `add_grab()` method.

`add_nr_events(*args) -> None`

add event types which will be ignored by the widget

The widget will neither handle the events nor send them to the children. The event types are saved in a `set`, therefore it is safe to add one event type multiple times.

`remove_nr_events(*args) -> None`

cancel the effects of `add_nr_events()`

It is safe to remove event type which has not been added before.

`add_ns_events(*args) -> None`

add event types which will not be sent to the children

Very similar to the `add_nr_events()` method. The only difference is that the events will be handled.

`remove_ns_events(*args) -> None`

cancel the effects of `add_ns_events()`

It is safe to remove event type which has not been added before.

`handle_event(event, _filter=True) -> None`

handle an event and send it to the children

`event` is an instance of `pygame.event.Event`. There are 2 main types of events: the ones generated by `pygame` (see the standard event types at `https://www.pygame.org/docs/ref/event.html`), others are generated by the widgets of `pygame_widgets`. We will not discuss the user-generated events here.

The events generated by widgets are always handled by the specific widget right after their creation (and they are send to the children), only then they are posted to the event queue. This has significant advantage in low latency, however it brings some problems. When the event queue is processed using

```
window.handle_events(*pygame_widgets.pygame.event.get())
```

or any equivalent code, the event propagates through the tree of widgets until it gets to the widget that created it. Then it would be handled twice by this widget and all its children, which is unwanted behaviour.

This problem is solved using signature. The event has an attribute `event.widget` which holds the reference to the widget it was created by. Then, any widget never handles event with its own signature.

Passing an argument `_filter=False` disables this filtering, allowing widget handle its own event. This is used internally in the library (specifically when the event is created - it has to be handled), but user should never call the method with non-default `_filter` argument.

Note: The filtering should be implemented differently. There should be method `_Master._handle_event()` which handles any event, and `handle_event()` should be only a wrapper with filtering.

```
handle_events(*events, _filter=True) -> None
```

handle all events and send them to the children

This method handles all events one-by-one using `handle_event`.

```
blit(rect=None, _update=True) -> None
```

redraw part of the display

This method uses the method `pygame.Surface.blit()`. It blits a part of `self.my_surf` to `self.surface`, ultimately leading to the change on the screen. The blitting area can be determined by the `rect` argument. When not passed, the return value of `self.surface.get_rect()` is used. The coordinates are relative to the topleft of `self.master_rect`.

This method is recursively called in all children colliding with `rect`. For optimization reasons is the `_update` argument set to `False`, although user should never overwrite the default value.

`pygame_widgets.Window`

Widget representing the whole window of the application. The absolute master of all widgets.

```
Window(size=(0, 0), flags=0, depth=0, **kwargs) -> Window instance
```

The object `Window` is used to control the basic functions of the application. Once an instance is created, the window appears on the screen and it is possible to create another widgets and display them inside. The object also manages event handling process and distributes the events to the other widgets.

The parameter `size` determines the resolution (and therefore the size) of the window. In case one of the dimensions is 0, this dimension is set to the corresponding

dimension of the screen resolution. Parameters `flags` and `depth` have the same meaning as the corresponding parameters of `pygame.display.set_mode()`. It is usually recommended not to set the latter, because `pygame` sets it to the best values for the system. The parameter `flags` adjusts the behaviour of the window.

The keyword arguments are used to set the window right after the initialization. It is meant only as simplification; the two following blocks of code have the exactly same effect:

```
window = pygame_widgets.Window(**kwargs)
```

```
window = pygame_widgets.Window()
window.set(**kwargs)
```

The description of all possible keyword arguments is listed in the documentation of the method `Window.set()`.

This class inherits the class `_Master`.

Warning: `pygame` allows creating only one window. In case user attempts to instance the class `Window` multiple times, the application will behave unpredictably.

`set(**kwargs) -> None`

set the attributes of the widget

All settable attributes of every widget can be changed using this method. Every widget has its own list of settable attributes, while all widgets inheriting the class `Window` support all arguments `Window` does.

For every attribute set, an event of type `E_WINDOW_ATTR` is published. The event has the following attributes: `name` (string containing the name of the changed attribute), `new` (the new value) and `old` (original value; for some attributes `None`).

`cursor`	the default appearance of cursor
`fps`	maximum frame rate (uses `pygame.time.Clock`). In case `fps==0`, frame rate is not limited.
`bg_color`	background color
`min_size`, `max_size`	determines the minimum and maximum size of the application window in case user has the ability to resize it (a flag `RESIZABLE` is enabled) or the program itself tries to change it
`title`	sets the title displayed in the heading of the window
`icontitle`	the title displayed on the main panel. In case it is set to `None`, the `title` is used.
`icon`	the icon displayed on the main panel
`size`	changes the resolution (and therefore size) of the window

`quit(code=0) -> None`

quit the application

Deinitializes all active widgets and then `pygame`. Then it calls `sys.exit(code)`.

`delete() -> None`

recursively deinitialize all active widgets

`update_display() -> None`

actualise the image on the monitor

Draws all changes that happened since the last call on the monitor. It also actualises the clock (see the documentation of `pygame.time.Clock.tick()`).

This method should be called in every cycle of the mainloop exactly once to assure the image on the monitor is up-to-date. In case there is a limit of maximum frame rate, `update_display()` always waits not to exceed it.

During the call of `pygame_widgets.new_loop(window)` with