This document provides an introduction to event-based programming, and to graphics programming in pygame.

**Event-based versus command-line interaction**

There are two fundamentally different ways to support a human user interacting with a program while it is running: (a) Command-line-only and (b) Event-based.

**(a) Command-line-only interaction.**

Command-line-only interaction works for software that will only be run on the command line, such as on a terminal or console. The user issues commands exclusively with the keyboard. There can be no mouse or touchscreen interaction.

A command-line-only program follows the following relatively simple structure.

(What follows is “pseudo-code”, code that summarizes commands that would be written for a computer, but without using a specific programming language. Read each line as if it were a command to the computer.)

```python
1    while (program_running == True)
2
3      prompt the user for input
4
5      wait for the user input (text)  # The computer waits here.
6
7      # Process the input.
8      if the user entered <text 1>:
9        do <action 1>
10     else if the user entered <text 2>:
11       do <action 2>
12     else if the user entered <Quit>:
13       program_running = False
14     # Loop back to the start.
```

Note where the program sits waiting, on line 5. The program spends the majority of its time sitting and waiting for user input on line 5. Once the user enters a command, the program blows through all the other lines of code to execute the command, and then goes back to waiting for the user. Until the user enters a command, the program does nothing but waits. You can easily trace the precise flow-of-control through the program. You can predict the exact order in which statements will be executed.
Here is an example of a command-line-only interaction from the user’s perspective. This shows a user changing the directory, and listing the contents of a directory, in Unix.

```bash
> cd /
> ls
Applications   Library
System         Users
>
```

Other examples of command-line-only interfaces include the Macintosh Terminal.app, the Windows command line, and DOS (the original Microsoft Disk Operating System).

**b) Event-based interaction.**

Event-based interaction is necessary for software that supports mouse-input or touchscreen-input. Event-based interaction can also support other events, such as keystrokes, or network activity. To support this arrival of events, at any time, from many possible sources, the program needs to be checking for different inputs in parallel (or at least in an interleaved manner). The program cannot sit on one line of code waiting for the user to hit <Enter>.

Event-based code is more complicated than command-line-only code. Event-based code needs to check for many different possible inputs, often from multiple input devices, all at the same time. It needs to be able to respond to events while also waiting and watching for new events to arrive.

The code typically looks something like the following. (This is pseudo-code.)

```python
# Top-level event loop.
while (program_running == True):
    wait for an event to occur
    dispatch the event to an event-handler # but come right back
    # without waiting for the event-handler to finish processing.

# Handle keystroke events in a separate thread.
if a key_hit_event occurred:
    if <key x> was pressed:
        do <action x>
    ...
else if <ESC> was pressed:   # The <ESC> key terminates the program.
    program_running = False

# Handle a GUI button-press event in a separate thread.
if a GUI_button_press event occurred:
    if <screen_button_x> was clicked:
        do <screen_button_x action>
    ...
# Clicking the "close window" button terminates execution.
else if <window_close_button> was clicked:
    program_running = False
```
It is important to note that the three above blocks of code all run (effectively) in parallel. The event loop can do the following, in this order:

1. Catch a keystroke.
2. Start the code that processes the keystroke.
3. Without waiting for the keystroke code to finish, resume the top-level loop.
4. Catch a mouse-click.
5. Start the code that processes the mouse-click.
6. Without waiting for the mouse-click code to finish, resume the top-level loop.

It is difficult to trace the precise flow-of-control through an event-based program because it is difficult to precisely predict when the processing of an event will start, how long the processing will take, and when the processing will be done. It is also difficult to predict how the processing of different events will be interleaved.

Programs that support event-based interaction accomplish the interleaving of multiple processes by using threads. A thread is a sequence of processing in a computer program that occurs in parallel with, and independently of, other threads in the same program. Because of the use of threads, we say that events are dispatched; that is, they are sent off to be taken care of on their own. It is not like a command-line-only interaction in which all of the code associated with an input is processed, in order, and then control is passed back to the input loop.

Many application frameworks (code and libraries for creating software) for building graphical user interfaces (GUIs) do not give the programmer access to the top-level event loop, but instead just permit the programmer to write code that responds to events, such as by listening for events with listeners (code that sits ready to respond to an event). If the programmer access to the top-level event loop, this also makes it hard to follow the precise flow of control such as with print statements or a debugger.

Note that, especially if the computer has only a single CPU, only one instruction can be executed at a time, and so two threads are not truly executed in parallel—with instructions from each thread firing at the exact same time—but are instead interleaved; that is, the CPU goes back and forth running instructions for one thread, then the other thread, then the one, and so on.

We next consider event-based programming in pygame.
Event-based programming in pygame

Pygame uses event-based programming, and reveals the top-level event loop within pygame. The loop looks like this:

```python
program_running = True
while program_running:
    for event in pygame.event.get():
        # This handles a couple ways to quit pygame.
        if event.type == pygame.QUIT:
            program_running = False
        # This handles keystrokes.
        elif event.type == pygame.KEYDOWN:
            # Process an <F> keystroke.
            if event.key == pygame.K_f:
                pass
            if event.key == pygame.K_ESCAPE:  # The escape key.
                program_running = False
                going = False
        elif event.type == event.key == pygame.K_ESCAPE:
            going = False
        # The event loop was ended. Quit the program.
        pygame.quit()
```

Do not “pause” the flow of control through the event loop.

It is important that you do not write code that pauses the flow of control through the event loop. The code could look like any of the following.

```python
DO NOT INCLUDE ANY OF THE FOLLOWING STATEMENTS IN YOUR PYGAME CODE:
while pygame.mixer.music.get_busy():
    pass
pygame.mixer.get_busy():
    pass
pygame.time.wait(1000)
pygame.time.delay(6000)
time.sleep(0.5)
```

Code written in pygame that processes events should, in general, not have any code that looks like any of the above. All of the code above, regardless of where it appears in your program, would pause the event loop, and prevent the program from responding to any new inputs from the user. The event loop needs to run continuously. The event loop should only interrupted to dispatch events; that is, to start events. The event loop should not be interrupted to process the
events if that processing would temporarily halt the continuous looping through the main event loop.

If the main event loop is paused, no new events are processed. The user's experience is that the system stops responding, and no longer responds to new user input. The computer appears broken. The operating system sometimes indicate that a program is not responding to new user inputs such as, on the Macintosh, by showing the spinning wait cursor, the “spinning beach ball”.

Sound files, for example, should be started and stopped with functions called from within the main event loop, but those functions should not wait until the sounds are done playing before returning control back to the main event loop.

Event-Based Programming in Pygame - Resources

*A Newbie Guide to pygame*
A good summary of some important concepts is at
https://www.pygame.org/docs/tut/newbieguide.html
The section on “Managing the event subsystem” discusses how the top-level event loop works in pygame, and relates directly to the above discussion on event-based programming.

If you are not building a game with animation, you can probably skip these sections:
• Dirty rect animation.
• Hardware surfaces are more trouble than they’re worth.
• Don’t bother with pixel-perfect collision detection.

*Eventlist.py*
This program provides a nice illustration of many of the events that occur in pygame, and provides an example of event-loop programming.
It is discussed briefly at
https://www.pygame.org/docs/ref/examples.html#pygame.examples.eventlist.main
and is available at
https://github.com/takluyver/pygame/tree/master/examples

Graphics Programming in Pygame - Resources

Pygame supports graphics programming, evidently using an approach and terminology common in graphics programming. Here are some good resources on the pygame website for learning graphics programming in pygame.

*The chimp.py tutorial*
The creator of pygame (Pete Shinners) created an extended tutorial to illustrate many of the key concepts in graphics programming in pygame. The tutorial is at
https://www.pygame.org/docs/tut/ChimpLineByLine.html
Important Graphics Programming Concepts

The above tutorials should help you to learn some important concepts such as …

`pygame.display.flip()` and `pygame.display.update()`

In graphics programming, it is often the case that visual elements are initially written to a buffer, and then later displayed on the actual screen. Sometimes an explicit function call is needed to send the contents of the buffer to the screen. The two pygame commands for accomplishing this are `pygame.display.flip()` and `pygame.display.update()`. Any time that you draw something to the display, such as with `pygame.Surface.blit()`, you also need to issue a `flip()` command for the new content to actually appear.

Conclusion

Event-based programming is fundamentally different from command-line-input, but most modern interfaces use event-based programming, and it is great to understand how it works. The pygame library offers a useful toolkit for cross-platform event-based programming, and also supports audio and visual media such that a range of different user interfaces can be built without learning a complex toolkit, and without being constrained to a specific set of interface controls.