Pygame - Dev Notes - Getting started

A few notes on getting started with Pygame and basic game development.

Contents

Intro

These notes offer a brief intro to working with Python and Pygame to help create our first video game with this development library.

What is Pygame?

Pygame is a powerful and useful set of modules to help develop and create games with Python. Further details may be found on the Pygame website at the following URL,

- Pygame

Game development

As we do for other languages and development, e.g. web stack based development, we can create a template file for starting our Pygame based projects. There are a number of ways to setup a template for such game based development.

imports

We'll start by importing the *pygame* module, which helps setup our template for use with required modules

```
# import modules for pygame template
import pygame
```

Pygame window defaults

We can then add some defaults for a *window* in Pygame, defining our variables as follows

```
# variables for game window and speed
winWidth = 800
```

```
winHeight = 600
FPS = 30
```

We're setting the default window size, and the frames per second for the game itself. In effect, how fast the game will update per second on each system. We can obviously update this value for each game's requirements.

Our game loop will then reflect the number of frames per second, which means it will now run itself 30 times each second. Effectively, each loop is set to 1/30 of a second.

Pygame initialise

The next thing to add to our Pygame template is the general initialisation for our game's initial settings.

We can start by initialising Pygame, and the sound mixer. The sound mixer allows us to play back sound at various points in our game.

Then, we can create our screen or window for the game, and add a brief caption for this window.

As we're defining the FPS for our game, we also need to define a clock. This clock helps us track how fast the game is going, and allows us to ensure that we're maintaining the correct FPS.

```
# initialise pygame settings and create game window
pygame.init()
pygame.mixer.init()
window = pygame.display.set_mode((winWidth, winHeight))
pygame.display.set_caption("game template")
clock = pygame.time.clock()
```

Game loop

We've now setup and initialised the basics for our template. However, we need to add a basic game loop to our Pygame template.

One of the key requirements for developing a game, including with Python and Pygame, is the creation of a game loop.

A *Game loop* is executed for every frame of the game, and the following three processes will happen:

- **processing inputs** (aka events)
  - responding to interaction from the player within the game - e.g. keyboard press, mouse, game controller...
  - listening for these events, and then responding accordingly in the game's logic
- **updating the game**
  - updating, modifying anything in the game that needs a change...graphics, music, interaction &c.
  - a character moving - need to work out where they might be moving &c.
  - characters, elements in the game collide - what happens when they collide? &c.
  - i.e. responding to changes in state and modifying a game...
- **rendering to the screen**
  - drawing modifications, updates, &c. to the screen
  - we've worked out what needs to change, we're now drawing (rendering) those changes

As part of the game's development, we may also need to consider how many times this *game loop* repeats. i.e. the frames per second that this loop repeats. FPS is important to ensure that the game is not running too fast or too slow.

However, specific FPS usage will often depend upon the type of game being developed.

As noted above, the standard pattern requires,

- processing inputs (events)
- updating the game
- rendering to the screen

add loop

For each game we develop, we'll need to add a game loop to control and manage this pattern. In effect, we're listening for inputs, events, then updating the game, and finally rendering any changes for the user.

So, we can add a standard `while` loop as our primary game loop. e.g.

```python
# boolean for active state of game
active = True
# create game loop
while active:
    # 'processing' inputs (events)
    # 'updating' the game
    # 'rendering' to the screen
```

This loop is following our pattern of listening and processing inputs, updating the game, and finally rendering or drawing the game to the display for the player.

The boolean `active` allows us to monitor the active state of the game loop. As long as the value is set to `True` it will keep running. By updating this value to `False` we can then exit this `while` loop, thereby avoiding the dreaded *infinite loop*. The use of this boolean value is one option for exiting the `while` loop. An alternative is also shown in the final code example of this document. This follows better practices for working with Pygame.

processing inputs - events

As the game is running, a player should be able to interact with the game window, such as clicking the exit button, or using one of the defined control options, perhaps a keyboard option.

However, if we consider the nature of a `while` loop we may see an issue with the underlying logic. In effect, what happens if a user clicks a button on the keyboard whilst the loop is either *updating* or *rendering*.

Obviously, we need to be able to listen and record all events for our game regardless of the current executed point in the `while` loop. If not, we would only be able to listen for events at the start of the loop, as part of the *processing* logic.

Thankfully, Pygame has a solution for this issue.

Pygame event tracking

Pygame is able to keep track of each requested event from one executed iteration of the *game loop* to the next. In effect, it will remember events as the game's `while` loop executes the *updating* and *rendering* logic for our game.

Then, as the `while` loop executes the *processing* logic, we're able to check if there have been any new events. For example, we can now add a simple `for` loop to check for each and every event that Pygame has saved, e.g.

```python
...
for event in pygame.event.get():
    ...
```

We can start by checking for an event registered as clicking on the exit button to close the current game window. e.g.

```python
...
for event in pygame.event.get():
    # check for window close click
    if event.type == pygame.QUIT:
        # update boolean for running
        active = False
```

So, we're checking for a saved event that simply indicates the user wants to close the current game window. We then update the value of the boolean for the active game, setting the value of the `active` variable to `False`. The game loop, our `while` loop, will now exit.

We can then add a call to quit Pygame at the end of our current Python file. e.g.

```
...
pygame.quit()
```

The game will now exit, and the Pygame window will close.

rendering - double buffering

As we start to render colours, lines, shapes &c. to our Pygame window, we need to be careful not to re-render everything for each update. Our game would become very resource intensive if we need to draw everything for each event and update in our game per *game loop*.

With this in mind, we can use an option known as **double buffering**.

In Pygame, this uses a concept of pre-drawing and then rendering as and when the drawing is ready to be viewed by the player.

For example, an artist is drawing a portrait of a customer sitting in front of them. When they are happy with the drawing, they simply turn or **flip** this drawing so the customer, the sitter, can see the finished portrait.

We can add this to our template as follows,

```
...
# flip our display to show the completed drawing
pygame.display.flip()
```

This **flip** must be the last call after drawing. If not, nothing will be displayed to the game's player.

monitor FPS

Our *game loop* may also need to monitor and maintain the defined setting for our game's FPS.

Currently, we've set this to run at 30 frames per second. We need to ensure this is monitored as part of our *game loop*, i.e. within the logic of our game's `while` loop.

```
...
# check game loop is active
while active:
    # monitor fps and keep game running at set speed
    clock.tick(FPS)
...
```

So, Pygame is now able to keep our game running at the defined frames per second. As the loop runs, it will always ensure that the loop executes the required 1/30 second.

Therefore, as long as the loop is able to *process*, *update*, and *render* within this defined time period, the game will run correctly. If not, and the *update* is taking too long, the game will end up running with lag, and appear jittery to the player. This is when we need to consider optimisation of code &c.

Finish the basic template

As we're only listening for the exit event on the game window, we don't currently have any game content to update.

So, our current template has set up a game window, and environment, to test initial setup and initialisation, and then allow a player to exit the game and window.

```
...
# quit the Pygame window, exiting the game
pygame.quit()
...
```

Another example template

We may create our initial template using many variations on the above patterns. For example, another template might be as follows,

```
# import modules for pygame template
import pygame, sys

# variables for pygame
winWidth = 800
winHeight = 600

# variables for commonly used colours
BLUE = (0, 0, 255)

# initialise pygame settings and create game window
pygame.init()
window = pygame.display.set_mode((winWidth, winHeight))
pygame.display.set_caption("game template")

# define game quit and program exit
def gameExit():
    pygame.quit()
    sys.exit()

# create game loop
while True:
    # 'processing' inputs (events)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            gameExit()
    # 'updating' the game

    # 'rendering' to the window
    window.fill(BLUE)
    # 'flip' display - always after drawing...
    pygame.display.flip()
```