Chapter 3 Building experiments with Pygame

Pygame is a Python wrapper for the widely used SDL multimedia library. Pygame is suitable for tasks like creating 2D graphics and monitoring keyboard, mouse, and gamepad events. It is the back-end library of many popular experiment creation tools like PsychoPy and SR Research Experiment Builder. It may seem a bit redundant to have a chapter on Pygame since we have covered the frequently used features of PsychoPy (which itself uses Pygame). However, while PsychoPy depends on about 60 different Python modules, Pygame requires no additional dependencies. As such, it is a lightweight choice for tasks involving simple 2D graphics.

Install Pygame

To install Pygame, execute the following command in a terminal (or command line) window.

```
pip install pygame
```

To test if Pygame is appropriately installed, start a Python shell, and enter the following command to see if you can import Pygame without issue.

```
import pygame
```

The Pygame library includes multiple modules, e.g., *display*, *draw*, *font*, *image*, *key*, *mixer*, *mouse*, *time*, etc. In the following sections, we will introduce some of the most frequently used modules through examples. Before we dive in, there are a few things to clarify.

Note that most Pygame modules require "initialization" before you can use their various functions and features. For instance, to play audio files and to present texts on-screen, we need to initialize the *mixer* and *font* modules.

```
pygame.mixer.init()
pygame.font.init()
```

One common practice is to call *pygame.init*() to initialize all modules at the beginning of each script. The downside of this approach is that Pygame may complain or crash if some modules are not available on a system (e.g., the *mixer* module on a Linux machine without ALSA sound driver).

The Pygame library also contains lots of constants; for instance, *pygame.K_z* is the ASCII code for letter "z" (i.e., 122). It can be cumbersome to refer to these constants with the "pygame." prefix. To avoid this, import all the "local" constants of the Pygame library at the beginning of the scripts.

```
>>> from pygame. locals import *
>>> K_z
122
```

Display

As with PsychoPy, we first need to open a window to draw various graphical elements and to detect keyboard and mouse events. Opening a window requires just one line of code, but there are some caveats. It is essential to understand that a window by its very nature is a surface. You can draw various geometric shapes on a window or place another surface (e.g., an image loaded into the memory) onto a window.

```
pygame.display.set_mode(size=(600, 400), flags=0, depth=0, display=0)
```

The size argument requires a tuple specifying the size of the window in pixels. If a window is open in full-screen mode and the specified window size is not one of the supported screen resolutions, Pygame will use the closet match. The depth argument is usually not set so that Pygame will default to the best and fastest color depth for the system. The argument that we need to specify is the flags, which determine what type of window to initialize. We can combine

multiple flags through a bitwise OR operation, by separating the options with the pipe ("|") character. For better timing performance, in most experimental scripts, we should supply the FULLSCREEN, DOUBLEBUF, and HWSURFACE flags to tap into the capacity of modern graphics cards (e.g., hardware acceleration).

```
win = pygame.display.set_mode((600, 400), HWSURFACE|DOUBLEBUF|FULLSCREEN)
```

The display module also comes with a handy tool for retrieving all supported screen resolutions, i.e., *pygame.display.list_modes()*. Use this command only after you have called *pygame.init()* or *pygame.display.init()*.

```
>>> pygame.display.list_modes()

[(2560, 1600), (2048, 1280), (1650, 1050), (1440, 900), (1280, 800), (1152,
720), (1024, 768), (800, 600), (840, 524), (640, 480)]
```

The other frequently used function is *pygame.display.get_surface()*, which returns a reference to (or a handler of) the currently active window. *pygame.display.get_surface()* is handy when defining custom drawing functions. In the pseudo-code below, instead of directly referring to the window surface we initialized, we refer to the surface returned by *pygame.display.get_surface()*. So, we can build functions that are not tied to a hardcoded window surface, i.e., *win* in the pseudo script below.

```
win = pygame.display.set_mode((600, 400))

def my_draw_circle(size, where):
    ''' draw a circle '''

    surf = pygame.display.get_surface()
    pygame.draw.circle(surf, color, position, size)
    pygame.display.flip()
```

For some visual psychophysics tasks, it is necessary to gamma correct the monitor (see Chapter 2 for a brief discussion on monitor gamma and Chapter 9 for instructions on calculating the gamma of a monitor). One can use the following command to set the gamma value for the RGB channels to linearize the video output.

```
pygame.display. set_gamma(red, green, blue)
```

The refresh rate of a monitor and its timing consistency is critical to psychophysics studies. The short script below shows a simple way of checking the refresh intervals, but it also serves as a short script that allows us to see how Pygame works. We have explained the first few lines already before. We import Pygame and all the local constants, initialize the modules and then open a full-screen window that enables the graphics card capacities of double buffering and hardware acceleration.

```
# Filename: display_demo.py

import pygame
from pygame.locals import *

pygame.init() # initialize Pygame modules

# open a window
win = pygame.display.set_mode((1024, 768), DOUBLEBUF|HWSURFACE|FULLSCREEN)

# create an empty list to save the monitor refresh intervals
intv = []

# flip the video buffer to make sure the first timestamp corresponds to a
retrace
pygame.display.flip()

# get the timestamp of the 'previous' screen retrace
t_before_flip = pygame.time.get_ticks()

# use a for-loop to flip the video buffer for 100 times
for i in range(100):
    # constantly switching the window color between black and white
    if i%2 == 0:
        win.fill((255, 255, 255))
    else:
        win.fill((0,0,0))
    pygame.display.flip() # flip the video buffer

    # get the timestamp of the 'current' screen retrace
    t_after_flip = pygame.time.get_ticks()
    flip_intv = t_after_flip - t_before_flip # get the refresh interval
```

```
    intv.append(flip_intv) # save the current refresh interval to a list
    t_before_flip = t_after_flip # reset the timestamp of the 'previous'
retrace

# print out the max, min and average refresh intervals
print('Max: {}, Min: {}, Mean: {}'.format(max(intv),
                                          min(intv),
                                          sum(intv)*1.0/len(intv)))
# quit pygame
pygame.quit()
```

Most modern graphics cards support double buffering. With double buffering, the computer draws the graphics in the back buffer. To show the graphics on the screen, we need to flip the back buffer to the front, with the *pygame.display.flip()* command.

```
pygame.display.flip()
```

The window itself is a Surface, so we can call *fill()* to change the color of the window. The other module used here is the *time* module, with which the *get_ticks()* function will return the number of milliseconds that have elapsed since the call to *pygame.init()*.

Running the above script in IDLE will give your graphics card a quick test. The test results for my Macbook Pro, which has an integrated Intel graphics card, were terrible. The results for my Windows PC, which has an Nvidia GeForce GTX graphics card and a 144 Hz BenQ monitor, were reasonably decent. The comparison here is about the graphics card and monitors, not the operating systems. We can achieve fairly reasonable timing on most Macs equipped with discrete graphics cards.

```
Max: 139, Min: 8, Mean: 19.58 # Results from my Mac
Max: 8, Min: 2, Mean: 6.89 # Results from my desktop PC
```

Events

To create an interactive application, one needs to handle the various events that are detected by Pygame. For instance, when you click the “close” button on a

window, a QUIT event will be generated; when you press a key down, a KEYDOWN event will be generated. There are several other types of events that are supported by Pygame, for instance, MOUSEMOTION (mouse motion) and JOYBUTTONDOWN (joystick button down).

Pygame queues the events, and there are various tools for handling the event queue. The event handling function that I prefer is *pygame.event.get()*, which grabs all the queued events and then empties the queue. This function will return a list of events that added to the queue since the last call of this function.

```
[<Event(12-Quit {})>]
[<Event(5-MouseButtonDown {'pos': (192, 124), 'button': 1, 'window':
None})>]
[<Event(2-KeyDown {'unicode': 'a', 'key': 97, 'mod': 0, 'scancode': 0,
'window': None})>]
[<Event(4-MouseMotion {'pos': (0, 185), 'rel': (0, 1), 'buttons': (0, 0, 0),
'window': None})>]
```

As shown in the shell output above, *pygame.event.get()* return event(s) in lists. Each event contains a unique code; for instance, the mouse motion event has a code of 4. These event codes are Pygame constants, typing "pygame.MOUSEMOTION" in a Python shell will return the type code 4.

```
>>> pygame.MOUSEMOTION
4
```

Of course, an event also contains information other than the type code. In the "MouseMotion" event shown above, the event properties that can be retrieved include '*pos*', '*rel*', and '*buttons*', which correspond to the current mouse position, (0, 185), the distance the mouse has moved since the previous MouseMotion event, (0, 1), and the status of the mouse buttons, (0, 0, 0). With this basic understanding of event handling in Pygame, we should be able to continuously retrieve the mouse position in a while loop, with the following sample code.

```
while True:
    ev_list = pygame.event.get()
    for ev in ev_list:
        if ev.type == MOUSEMOTION:
```

```
            print(ev.pos)
```

We first call *pygame.event.get()*, which gives us a list of events that we can loop over to see if there is a MOUSEMOTION event. If so, we print out the mouse "pos" stored in this event. With a similar method, we can, of course, retrieve key presses and other events, as shown in the sample example below.

```
# Filename: event_demo.py

import pygame
from pygame.locals import *

# initialize the modules and open a window
pygame.init()
scn = pygame.display.set_mode((640, 480))

while True:
    ev_list = pygame.event.get()
    for ev in ev_list:
        # mouse motion
        if ev.type == MOUSEMOTION:
            print(ev.pos)
        if ev.type == MOUSEBUTTONDOWN:
            print(ev.button)
        # key down
        if ev.type == KEYDOWN:
            print(ev.key)
        # close window button
        if ev.type == QUIT:
            pygame.quit()
```

The Pygame library also has *key* and *mouse* modules for handling keyboard and mouse inputs, but the event module is usually sufficient for most experimental purposes. For further information, please see the official Pygame documentation. [1]

[1] Pete Shinner has a nice blog on input handling in Pygame, https://www.pygame.org/ftp/contrib/input.html.

Draw

For simple drawings, one can use either the *draw* module or the *gfxdraw* module. The quality of geometric shapes created with the *draw* module is generally low but should be good enough for tasks requiring simple graphics, like the Posner cueing task presented at the end of this chapter. The *gfxdraw* module will give better results, but it is an experimental module and may be subject to changes in the future. Here we will present a simple example to show the drawing capability of the *draw* module. Every time the mouse is clicked, we store the mouse position into a list (of points). When the list has more than three points, we draw a polygon and use filled circles to mark the vertices.

```
# Filename: draw_demo.py

import pygame, sys
from pygame.locals import *

pygame.init() # initialize pygame

# open a window
scn = pygame.display.set_mode((640, 480))

# create an empty list to store clicked screen positions
points = []

while True:
    # poll all pygame events
    for ev in pygame.event.get():
        # quit pygame and Python if the "close" button is clicked
        if ev.type == QUIT:
            pygame.quit()
            sys.exit()

        # append the current mouse position to the list when
        if ev.type == MOUSEBUTTONDOWN:
            points.append(ev.pos)

    # clear the screen
    scn.fill((255,255,255))

    # draw a polygon after three mouse clicks
```

```
    if len(points) >= 3:
        pygame.draw.polygon(scn, (0,255,0), points)

    # show the screen locations that has been clicked
    for point in points:
        pygame.draw.circle(scn, (0,0,255), point, 10)

    # flip the video buffer to show the drawings
pygame.display.flip()
```

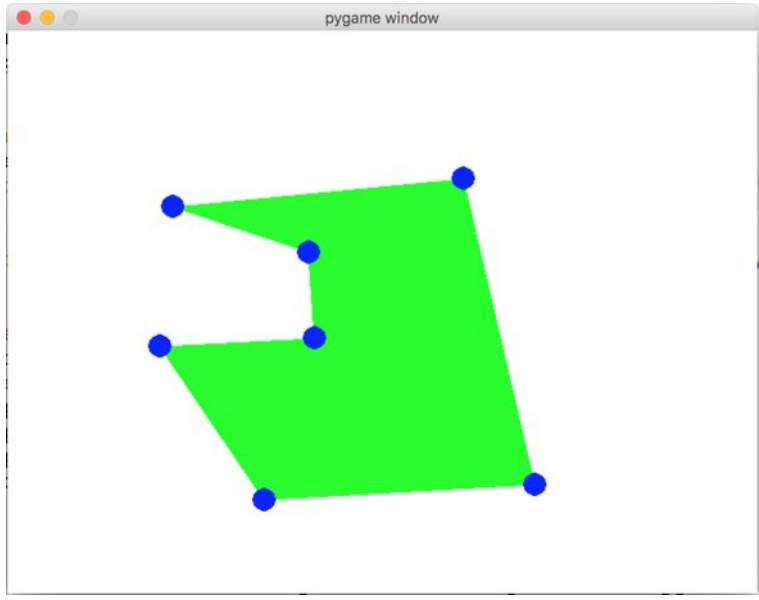

Figure 3-1 *Draw a polygon and show the vertices in Pygame.*

Figure 3-1 shows the result of the example code presented above. You can close the window by clicking the "Close" window button. Clicking this button, however, will also throw an error message in the shell because calling *pygame.quit()* will un-initialize all Pygame modules, causing the various drawing functions in the while loop to crash. To more elegantly terminate the script, call sys *.exit()* to exit the Python interpreter altogether.

Text

With Pygame, we need to initialize a font object to render text on the screen. To create a font object, call *pygame.font.SysFont()* to use system fonts or call *pygame.font.Font()* to use a custom TrueType[2] font file instead. For cross-platform applications, you may prefer *pygame.font.Font()* as the font you choose to use may not be available on all platforms, and the name of a font may differ across platforms. You can also specify a list of system fonts, so Pygame will exhaust these fonts before it reverts to the "default" font, which can be revealed by *pygame.font.get_default_font()*. To list all fonts available on your OS, call *pygame.font.get_fonts()*.

```
>>> pygame.font.get_default_font()
'freesansbold.ttf'
>>> pygame.font.get_fonts()
['cochin', 'iowanoldstyle', 'avenir', 'nanumgothic',
'bitstreamverasansmono',
...
'bodoniornaments', 'weibeitc', 'libiansc', 'plantagenetcherokee']
```

Here is a short script showing how to render text in Pygame. One can enable or disable underlining and bolding with *set_underline()* and *set_bold()*. The method *set_italic()*, however, only enables "fake rendering of italic text," which skews the font that "doesn't look good on many font types" (see the Pygame documentation). In the example script, we first initialize a font object; then, we call *size()* to estimate the size of the text surface after "Hello, World!" has been rendered. Rendering text will give us a surface, which can be presented (blitted) on another surface (e.g., a Pygame window). After blitting the text surface on the Pygame window, we flip the video buffer to show the texts on screen. If you need to terminate the script, press a key.

```
# Filename: text_demo.py

import pygame, sys
pygame.init()
```

[2] TrueType is a font standard developed by Apple and Microsoft in the late 1980s. It is the most common font format on MacOS and Windows.

```
# open a window
win = pygame.display.set_mode((300,200))

# create a font object and enable 'underline'
fnt = pygame.font.SysFont('arial', 32, bold=True, italic=True)
fnt.set_underline(True)

# size() estimates the width and height of the rendered text surface
w, h = fnt.size('Hello, World!')
print(w,h)

# render the text to get a surface
win.fill((0, 0, 0))
text_surf = fnt.render('Hello, World!', True, (255,0,0))

# show(blit) the text surface at the window center
win.blit(text_surf, (150-w/2,100-h/2))
pygame.display.flip()

# show the text until a key is pressed
while True:
    for ev in pygame.event.get():
        if ev.type == pygame.KEYUP:
            pygame.quit()
            sys.exit()
```

Note that the *font* module does not support text wrapping, so there is no easy solution for presenting multiline texts. Nevertheless, you can always use *size()* to figure out the size of each word in the texts to determine which word will reach the edge of the window and when to render a new line.

Image and sound

The *image* module allows users to load and save images, and to convert images into formats that are compatible with other tools. One can load pictures of frequently used formats (e.g., BMP, PNG, JPEG, etc.). The *pygame.image.load()* function will load an image and return a Pygame surface that contains the same color format as the image file. It is usually preferable also to call *convert()* to convert the image surface to match the pixel format of

the display surface. To draw the image on the display surface, call *blit()* and specify the target position.

```
win = pygame.display.set_mode((800, 600)) # open a window

img = pygame.image.load('test.png').convert() # load an image
win.blit(img, (0,0)) # blit the image to the top-left corner of the window
```

We need to initialize the *mixer* module before we use it to play a sound file. There are parameters that one can customize when initializing the mixer module, e.g., the frequency (sample rate), which should match the audio file you would like to playback. To do so, you need to call *pygame.mixer.pre_init()* before calling *pygame.init()*. The parameters for *pygame.mixer.pre_init()* include *frequency* (sample rate), *size* (8- or 16-bit encoding), *stereo* (1 for mono and 2 for stereo), and *buffer size* (number of buffered samples). With Pygame 2.0, the default buffer size is changed to 512 to reduce playback latency, and the default frequency is 22050 Hz.

To load and play a sound file (in WAV or OGG format), we first create a Sound object with *pygame.mixer.Sound()*. Then, call the Sound object methods, which include *play(), stop(), fadeout(),* etc. to control the playback of the audio file.

```
# create a Sound object
snd = pygame.mixer.Sound('win.wav')

# play the sound file repeatedly for 5-sec
snd.play(-1, 5000)
```

Presenting auditory stimuli can be a complicated matter. For precise playback timing, commercial software like Experiment Builder and E-Prime usually requires an ASIO-compatible sound card. Unfortunately, Pygame does not ASIO yet.

A real example: Posner cueing task

As noted, Pygame is perfect for simple experimental tasks, like the classic Posner cueing task (Posner, 1980). In the example below, I will use the task procedure of a recent publication by Fu et al. (2019), in which we examined if

dyslexic children are sluggish in shifting spatial attention. The design of the task is relatively straightforward. Following an uninformative peripheral cue (an empty box), a visual target (bright disk) appears on the screen. The target could appear at the cued location or a distance-matched location in the opposite visual field. Participants are required to make a speeded response to the target, and the inter-stimulus interval (ISI) between the cue and the target was manipulated to reveal the time course of attention orienting.

Figure 3-2 illustrates the sequence of events that occur in a single trial. A fixation appears in the central place holder for 1000 ms; then, one of the peripheral placeholder flashes (cue) briefly. After an ISI of 0, 100, 300, or 700 ms (randomized within blocks of trials), the target appears in either the cued peripheral placeholder or the uncued one. This experiment also monitored eye movements, but the relevant code is not included here for simplicity.

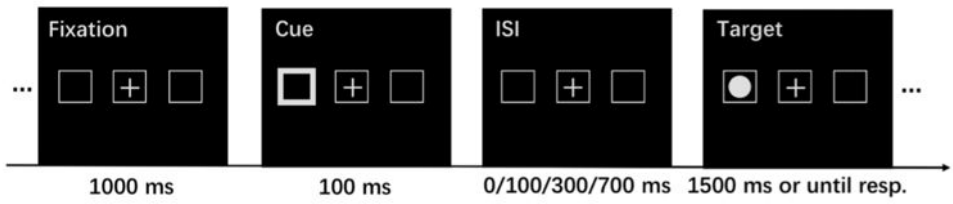

Figure 3-2 *The Posner cueing task used in Fu et al. (2019).*

The full-length script for this task is listed below. The structure of the script is similar to that of the PsychoPy example presented in the previous chapter.

```
# Filename: posner_cueing.py

import random, pygame, sys
from pygame import display, draw, Rect, time, event, key
from pygame.locals import *

# constants
sz = 90 # size of the placeholder
pos = {'left':(212,384), 'center':(512,384),'right':(812,384)}
colors = {'gray':(128,128,128),'white':(255,255,255),'black':(0,0,0)}

# list of all unique trials, [cue_pos, tar_pos, isi, cueing, cor_key]
```

```
trials = []
for cue_pos in ['left', 'right']:
    for tar_pos in ['left', 'right']:
        for isi in [0, 100, 300, 700]:
            if cue_pos == tar_pos:
                cueing = 'cued'
            else:
                cueing = 'uncued'
            if tar_pos == 'left':
                cor_key = 'z'
            else:
                cor_key = '/'
            trials.append([cue_pos, tar_pos, isi, cueing, cor_key])

def draw_frame(frame, trial_pars):
    ''' Draw the possible screens.

    frame -- which frame to draw, e.g., 'fix','cue', 'target'
    trial_pars -- parameters, [cue_pos, tar_pos, isti, cueing, cor_key]'''

    # unpack the trial parameters
    cue_pos, tar_pos, itiisi, cueing, cor_key = trial_pars

    # clear the screen and fill it with black color
    win.fill(colors['black'])

    # the fixation cross and the place holders are always shown visibly on
all screens
    # 'pos' is a dictionary, in which the key and value pairs can be
retrieved one-by-one in a for-loop
    for key, (x, y) in pos.items():
        # draw the place holder
        draw.rect(win, colors['gray'], Rect(x-sz/2, y-sz/2, sz, sz), 1)

        # draw the cross with two lines
        if key == 'center':
            draw.line(win, colors['gray'], (x-20,y), (x+20,y), 3)
            draw.line(win, colors['gray'], (x,y-20), (x,y+20), 3)

    # draw the fixation screen (do nothing because nothing needs to change)
    if frame == 'fix':
        pass

    # draw the cue (a bright box--a Rect)
```

```
    if frame == 'cue':
        c_x, c_y = pos[cue_pos] # coordinates of the cue
        draw.rect(win, colors['white'], Rect(c_x-sz/2, c_y-sz/2, sz, sz), 5)

    # draw the target (a filled disk)
    if frame == 'target':
        draw.circle(win, colors['white'], pos[tar_pos], 20)

    display.flip()

def run_trial(trial_pars, subj_info, data_file):
    ''' Run a single trial.

    trial_pars -- a list specifying trial parameters, [cue_pos, tar_pos,
isi, cueing, cor_key]
    subj_info -- info about the subject [id, name, age]
    data_file -- an open file to save the trial data.'''

    # show the fixation then wait for 1000 ms
    draw_frame('fix', trial_pars)
    time.wait(1000)

    # show the cue for 100 ms
    draw_frame('cue', trial_pars)
    time.wait(100)

    # ISI
    draw_frame('fix', trial_pars)
    time.wait(trial_pars[2])

    # show the target and register a keypress response
    draw_frame('target', trial_pars)
    t_tar_onset = time.get_ticks()
    t_tar_resp = -1 # response time
    t_tar_key = -1 # key pressed

    # check for key presses
    time_out = False
    got_key = False
    event.clear() # clear bufferred events, if there is any
    while not (time_out or got_key):
        if time.get_ticks() - t_tar_onset > 1500:
            time_out = True
```

```
        # check if any key has been pressed
        for ev in event.get():
            if ev.type == KEYDOWN:
                print(ev.key)
                if ev.key in [K_z, K_SLASH]:
                    t_tar_resp = time.get_ticks()
                    t_tar_key = key.name(ev.key)
                    got_key = True

    # write data to file
    trial_data = subj_info + trial_pars + [t_tar_onset, t_tar_resp,
t_tar_key]
    trial_data = map(str, trial_data)
    data_file.write(','.join(trial_data) + '\n')

    # ITI (inter-trial_interval)
    draw_frame('fix', trial_pars)
    time.wait(1500)

# -- real experiment starts from here --
# get subject info from the Python shell
subj_id = input('Subject ID (e.g., 01): ')
subj_age = input('Subject Age: ')
subj_info = [subj_id, subj_age]

# open a data file
d_file = open('d_{}.csv'.format(subj_info[0]), 'w')

# open a window, add the FULLSCREEN flag for better timing precision
win = display.set_mode((1024,768), HWSURFACE|DOUBLEBUF)

# randomly shuffle the trial list and test them one by one
random.shuffle(trials)
for pars in trials:
    run_trial(pars, subj_info, d_file)

# close the data files and quit the program
d_file.close()
pygame.quit()
sys.exit()
```

The script starts with importing the necessary Pygame modules and defining constants, e.g., the size and positions of the placeholders, and the colors used in the task (gray, white, and black).

```
# constants
sz = 90 # size of the placeholder
pos = {'left':(212,384), 'center':(512,384),'right':(812,384)}
colors = {'gray':(128,128,128),'white':(255,255,255),'black':(0,0,0)}
```

The simple trial control approach detailed in the previous chapter was used to implement this task. For a given trial, we need to manipulate the cue and target location, the cue-target ISI, and of course, we need to know which key is the correct one to press. Putting these variables in a Python list should give you something like *[cue_pos, tar_pos, ITI, cor_key]*. With nested *for*-loops, the following lines of code put all possible unique trials in a list. Note that I added another variable *cueing,* which can be "cued" (target in the cued box) or "uncued" (target in the uncued box), to the list. This variable will ease the task of deriving a cueing effect (cued - uncued) during data analysis.

```
# list of all unique trials, [cue_pos, tar_pos, isi, cueing, cor_key]
trials = []
for cue_pos in ['left', 'right']:
    for tar_pos in ['left', 'right']:
        for isi in [0, 100, 300, 700]:
            if cue_pos == tar_pos:
                cueing = 'cued'
            else:
                cueing = 'uncued'
            if tar_pos == 'left':
                cor_key = 'z'
            else:
                cor_key = '/'
            trials.append([cue_pos, tar_pos, isi, cueing, cor_key])
```

As shown in Figure 3-2, the task involves the drawing of multiple screens, i.e., a fixation-screen that contains a fixation cross and three placeholders (empty boxes), a cue-screen that includes the same elements but brightening one of the placeholders, and a target-screen with the fixation cross, the place holders, and the target (a bright disk). Here we define a simple function that we can use to draw all the screens. This function will take two arguments, *frame* and *trial_pars*. The argument *frame* is a label for the three types of the screen; it

could be ‘fix’, ‘cue’ or ‘target’. The argument *trial_pars* is a list that controls the behavior of each trial, e.g., where the target would appear.

```
def draw_frame(frame, trial_pars):
    ''' Draw the possible screens.

    frame -- which frame to draw, e.g., 'fix','cue', 'target'
    trial_pars -- parameters, [cue_pos, tar_pos, isi, cueing, cor_key]'''

    # unpack the trial parameters
    cue_pos, tar_pos, itiisi, cueing, cor_key = trial_pars

    # clear the screen and fill it with black color
    win.fill(colors['black'])

    # the fixation cross and the place holders are always visible on all
screens
    # 'pos' is a dictionary, in which the key and value pairs can be
retrieved one-by-one in a for-loop
    for key, (x, y) in pos.items():
        # draw the place holder
        draw.rect(win, colors['gray'], Rect(x-sz/2, y-sz/2, sz, sz), 1)

        # draw the cross with two lines
        if key == 'center':
            draw.line(win, colors['gray'], (x-20,y), (x+20,y), 3)
            draw.line(win, colors['gray'], (x,y-20), (x,y+20), 3)

    # draw the fixation screen (do nothing because nothing needs to change)
    if frame == 'fix':
        pass

    # draw the cue (a bright box--a Rect)
    if frame == 'cue':
        c_x, c_y = pos[cue_pos] # coordinates of the cue
        draw.rect(win, colors['white'], Rect(c_x-sz/2, c_y-sz/2, sz, sz), 5)

    # draw the target (a filled disk)
    if frame == 'target':
        draw.circle(win, colors['white'], pos[tar_pos], 20)

    display.flip()
```

One notable departure from the PsychoPy example is that we need to write a short routine to monitor keyboard events. After the target appears on the screen, we get the current timestamp and clear the event buffer; then, we use a while loop to wait for a keypress or until 1500 ms have elapsed. In PsychoPy, all we need is to call the *waitKeys()* function from the *event* or the *hardware* module.

```
# show the target and register a keypress response
draw_frame('target', trial_pars)
t_tar_onset = time.get_ticks()
t_tar_resp = -1 # response time
t_tar_key = -1 # key pressed

# check for key presses
time_out = False
got_key = False
event.clear() # clear buffered events, if there is any
while not (time_out or got_key):
    if time.get_ticks() - t_tar_onset > 1500:
        time_out = True

    # check if any key has been pressed
    for ev in event.get():
        if ev.type == KEYDOWN:
            print(ev.key)
            if ev.key in [K_z, K_SLASH]:
                t_tar_resp = time.get_ticks()
                t_tar_key = key.name(ev.key)
                got_key = True
```

This chapter introduces the frequently used modules of Pygame and presents a complete example illustrating the various functions of Pygame one can use in experimental scripts. The Pygame library is designed for computer programmers, not for experimental psychologists. I bet you may have developed a sense that scripts in Pygame tend to be longer and less straightforward compared to scripts in PsychoPy. Nevertheless, Pygame is still a decent multimedia library that one can use for various experimental tasks, for instance, a visual distractor task in which a visual target (of a saccade) is accompanied by a salient visual distractor (for an example script, see Chapter 7). Learning this library will help you to understand computer graphics and programming in

general. For readers who would like to learn more about this library, I would recommend the book by Will McGugan.

McGugan, W. (2007). Beginning Game Development with Python and Pygame: From Novice to Professional (First edition). Apress.