

## Introduction to Pygame

Pygame is an available add-on module to Python that provides tools for writing games and working with multimedia. It is a free module that is not part of the standard distribution of Python, but can be downloaded and installed on any platform for free:

- Visit <http://pygame.org/download.shtml>
- Install version 1.9.1 for whatever platform and version of Python you use at home (likely 2.6 or 2.5), but it is also available for the newer Python 3.x versions.
- In Wing, you can test that the install worked properly by typing in the Python shell `"import pygame"` (no quotes). If you get no errors, you are all set!

## A Basic 'Game'

The games we will write will all have the same basic framework. This means that we will establish a basic template/structure for our games, and then fill in each component. If you wish to use Pygame in ICS2/3, you can provide the bulk this framework, and students would be able to work with other more specific components (conditional event handling, pixel editing, etc).

### **Time:**

The two biggest differences you will have to adapt to from past programs you have written are adding a sense of ***time***, and learning to respond to different ***events*** in the game.

Games are in essence a loop that checks the status of the environment many times a second. This loop is often referred to as the ***game loop***, *main loop*, or *animation loop*. Because of the constantly changing environment, games have to process things faster than most other programs out there. The speed at which the game loop repeats is called the ***frame rate***, and is measured in terms of the number of refreshes per second. A frame rate of 30 is a pretty good starting point. This is the sense of time. The higher the frame rate, the smoother the game will look, but the greater burden on the processor. If you have ever had a game 'lag' or slow down, this may be why. Finding the right frame rate requires a balance of performance and processor demand.

### **Space:**

Your games will take place in a window. This window is made up of screen dots called ***pixels***. The number of pixels used determines the ***resolution***. As the resolution increases, there is a lot more pixel-processing that takes place. A window that is 400 pixels wide and 300 pixels high has 120000 total pixels to process. A window that is twice the width and height (800 x 600) has 4-times the number of pixels and 480000 (just under half a million!). If the frame rate was 30, then every second you could be updating over 14-million pixels... clearly the resolution is an important consideration.

In general, the window is like a Cartesian Plane, but the origin (0, 0) is the top-left corner, and the x and y coordinates increase to the right and down. The bottom-right pixel in the window would be at (width-1, height-1). Beyond that, we will not worry too much at this point about how the computer manages the pixels and updates a video display. If you have heard of DirectX or OpenGL, you may know a bit more already. Pygame takes care of all of that!

### **Execution:**

When executing your programs from Wing right now, you will find problems when you try to quit a program that you ran using the RUN button. As always, it is better to use the DEBUG button from Wing, and you won't have any extra worries (other than the bugs you caused yourself!).

The next page contains a first program for you to try *now*. Do not type in all the comments.

## Your First Pygame

"""

*A basic program establishing a simple framework for future games.*

"""

### **#Import & initialize the pygame module**

```
import pygame
from pygame.locals import * #This is not necessary, but makes constants easier to access
pygame.init()
```

### **#Set-up the main display window and the background**

```
size = (640, 480) #<-- that is a tuple (just like a list) for width & height
screen = pygame.display.set_mode(size) #<-- screen is now a Surface type object
pygame.display.set_caption("Yeah, Pygame!") #<-- caption appears in the title bar
```

```
background = pygame.Surface(size) #<-- like display, but creates a Surface object from scratch
background = background.convert() #<-- creates a copy of the Surface with a standard (faster)
# colour format. These two lines can be combined.
background.fill((0, 0, 255)) #<-- fills Surface with colour using a tuple (red, green, blue).
# See http://en.wikipedia.org/wiki/RGB\_color\_model for RGB info
# See http://en.wikipedia.org/wiki/List\_of\_colors for colours
```

### **#The game loop**

```
clock = pygame.time.Clock() #<-- used to control the frame rate
keep_going = True #<-- a 'flag' variable for the game loop condition
```

```
while keep_going:
```

```
    clock.tick(30) #<-- Set a constant frame rate, argument is frames per second
```

#### **#Handle any events in the current frame**

```
for ev in pygame.event.get(): #<-- returns a list of all Events in this frame
    # and this loops over each event, acting accordingly
```

```
#Events are objects with a type instance variable (an int, linked to pygame constants).
#These types could be a certain key pressed, a mouse moved, or even a guitar strum!
#These event types are mapped to constants in the pygame class.
#You can see them all with help(pygame).
```

```
if ev.type == QUIT: #<-- this special event type happens when the window is closed
    keep_going = False
```

```
elif ev.type == KEYDOWN: #<-- if it was a keyboard press
    if ev.key == K_r: #<-- keyboard events have a key property
        background.fill((255, 0, 0)) #<-- fill the background red
        pygame.display.set_caption("RED!") #<-- update the caption
    elif ev.key == K_g:
        background.fill((0, 255, 0))
        pygame.display.set_caption("GREEN!")
    elif ev.key == K_b:
        background.fill((0, 0, 255))
        pygame.display.set_caption("BLUE!")
    else:
        key_label = ev.unicode #<-- gets the unicode property value from the event
        pygame.display.set_caption(str(key_label)*10) #<-- update the caption
```

#### **#Update and refresh the display to end this frame**

```
screen.blit(background, (0, 0)) #<-- 'blit' means to copy one Surface to another
# Here, we copy the background onto the screen Surface
pygame.display.flip() #<-- refresh the display
```

\*\*\* Without all the comments, this is only about 40 lines.

## Playing with Colours

In the previous example, colours were represented as 3-part *tuples*. A tuple is essentially the same thing as a list, except it is *immutable*. You can replace the value in a tuple variable with a new tuple, but you cannot modify the elements within a tuple itself. You could instead use `lists` and Pygame will handle them accordingly, but tuples are returned by so many Pygame functions that it is worth while understanding what they are. Note that a tuple was also used at the start to store the size of the screen & background (640, 480), and at the end when updating the display starting at (0,0).

A colour (or 'color', as pygame uses American spelling) tuple is in the form (R, G, B), where each is an int value between 0 and 255. 0 means the complete absence of that element of the colour, and 255 is the greatest amount of that element of the colour. The colour (0,0,0) is black, while (255,255,255) is white. In truth, even though you can use tuples, pygame converts them behind the scenes to a `Color` type (`pygame.Color`) that has its own small set of methods. These `Color` objects can be compared with tuples using relational operators as if they were the same type. In addition, a colour tuple can have a 4<sup>th</sup> element representing the 'alpha' value, or 'transparency'. We will not discuss this just yet.

See the following links for more general info on RGB colours:

- [http://en.wikipedia.org/wiki/RGB\\_color\\_model](http://en.wikipedia.org/wiki/RGB_color_model)
- [http://en.wikipedia.org/wiki/List\\_of\\_colors](http://en.wikipedia.org/wiki/List_of_colors)

The previous example program responded to keyboard events by changing the background colour to pure red, green, or blue when the r, g, or b keys were pressed.

We could make a more interesting effect when the window is closed by fading out the background colour to black. When the window is closed, the `QUIT` event occurs, and we handled that by changing our while loop flag variable, causing our game loop to terminate. We can add the following code below the game loop (comments can be left out):

```
#Fade the background colour to black once game loop is done
colour = background.get_at( (0,0) ) #get the colour on the background Surface
while colour != (0,0,0):
    #decrease the r/g/b by 1, but don't allow them to go below 0
    colour = (max(colour[0]-1,0), max(colour[1]-1,0), max(colour[2]-1,0))
    background.fill(colour) #refill the background with the new colour
    screen.blit(background, (0, 0))
    pygame.display.flip()
```

Pygame also has a collection of 657 named colours, accessible through a built-in dictionary:

```
colours = pygame.color.THECOLORS
for colour in colours.keys():
    print colour, colours[colour]
```

Colours can now be used based on their name as shown below:

```
background.fill(pygame.color.THECOLORS["burlywood"])
```

*\*\*\*Note that 657 named colours is a tiny subset of the 16 million possible colours using the RGB model.*

Simply working with basic colours gives a number of playful exercises, including random numbers, handling the upper and lower bounds of the colours, exploring interesting colours such as shades of grey (all r/g/b values are equal), looking at colours and their negatives, etc.

## Playing with Events

In each frame, any sort of action on an input device will create an event. The command `pygame.event.get()` returns a list of the events in the current frame (as Event objects). This list can then be iterated over, and each event type can be detected and responded to accordingly. The previous example started by handling the basic `QUIT` event.

If the type of event is a `KEYDOWN` event, then you can get the `key` property. This is an `int`, which maps to a pygame constant (you can see these using `help(pygame)` ).

Holding a key down does not cause multiple events. To do so you will need to set-up the repeat interval:

```
pygame.key.set_repeat(delay, interval)
```

*#delay is milliseconds before first repeat, and interval is ms between repeats*

In the table to the right is a summary of the different types of events, and their corresponding properties. As you can see, it is important that you first determine which type of event you have before attempting to access its properties. If you attempt to access the 'key' property of a mouse event, you will get an error.

Many of these you will not use, but some we will see later with mouse handling, and using different joysticks.

Working with basic events can be linked to several concepts, including a good example of the use of constants, selection statements, logical operators, and nesting.

Event Type	Event Properties
QUIT	N/A
ACTIVEEVENT	gain, state
KEYDOWN	unicode, key, mod
KEYUP	key, mod
MOUSEMOTION	pos, rel, buttons
MOUSEBUTTONUP	pos, button
MOUSEBUTTONDOWN	pos, button
JOYAXISMOTION	joy, axis, value
JOYBALLMOTION	joy, ball, rel
JOYHATMOTION	joy, hat, value
JOYBUTTONUP	joy, button
JOYBUTTONDOWN	joy, button
VIDEORESIZE	size, w, h
VIDEOEXPOSE	N/A
USEREVENT	code

### **Another Way To Check Input:**

An alternative to handling input from events is to check the status of input devices directly. This allows you to check the status of multiple keys simultaneously without having to set-up extra variables. For the keyboard, the command `pygame.key.get_pressed()` will return a large tuple (323 values) full of `bool` values representing the state of each key. The indices in the tuple match the values of the pygame key constants. For example:

```
pygame.event.pump() #This is needed once per game loop iteration
                    #when not using event handling as above
keys = pygame.key.get_pressed() #this gets the tuple of bools for each key
if keys[K_ESCAPE]: #this checks the status of the ESC key, True/False
    keep_going = False
if keys[K_r]: #this checks the status of the r-key, True/False
    background.fill((255, 0, 0))
    pygame.display.set_caption("RED!")
elif keys[K_g]:
    background.fill((0, 255, 0))
    pygame.display.set_caption("GREEN!")
elif keys[K_b]:
    background.fill((0, 0, 255))
    pygame.display.set_caption("BLUE!")
```

There are some drawbacks to using this approach. The status of the keys are checked when the `get_pressed()` line executes. It is possible that a key is pressed and released so quickly that it is not down when the call is executed. It is also not possible to tell the order in which keys were pressed if multiple keys are down. The code above will always give precedence to the R key over the G or B, since it is checked first. If the 'elifs' were changed to 'ifs', then the precedence would be reversed.

## Surfaces

The `Surface` object is probably the most important element in all of Pygame. You have already seen a couple – the special one used for the base screen, and the one we used as a coloured background. These Surfaces are rectangular ‘layers’, like sheets of paper, which can be placed on top of each other, resized, rotated, coloured, etc. Let’s look at an example using multiple Surfaces:

```
import pygame
from pygame.locals import *
pygame.init()

full_size = (640, 480) #full screen size
bar_size = (640, 80) #size of each colour bar
screen = pygame.display.set_mode(full_size)

background = pygame.Surface(full_size).convert()
bg_colour = (0, 0, 0) #it helps to have the colour as a separate variable
background.fill(bg_colour)
pygame.display.set_caption("Colour: " + str(bg_colour)) #our caption will change

#Create the three colour bars
red_bar = pygame.Surface(bar_size).convert()
red_bar.fill((255, 0, 0))
green_bar = pygame.Surface(bar_size).convert()
green_bar.fill((0, 255, 0))
blue_bar = pygame.Surface(bar_size).convert()
blue_bar.fill((0, 0, 255))

clock = pygame.time.Clock()
keep_going = True
while keep_going:
    clock.tick(30)

    for ev in pygame.event.get():
        if ev.type == QUIT:
            keep_going = False
        elif ev.type == MOUSEBUTTONDOWN: #A mouse click!
            x = ev.pos[0] #the MOUSEBUTTONDOWN event has a position property
            y = ev.pos[1] #that is an (x, y) tuple

            #Determine the bar that was clicked by checking the y-coordinate
            if y >= 0 and y < 80: #we hit the red bar
                #Convert the x-coord to a % of the width and amount of red element
                #then adjust that element of the bg_colour, keeping the other 2.
                red = (1.0*x/(bar_size[0]-1))*255
                bg_colour = (int(red+.5), bg_colour[1], bg_colour[2])
            elif y >= 80 and y < 160: #we hit the green bar
                green = (1.0*x/(bar_size[0]-1))*255
                bg_colour = (bg_colour[0], int(green+.5), bg_colour[2])
            elif y >= 160 and y < 240: #we hit the blue bar
                blue = (1.0*x/(bar_size[0]-1))*255
                bg_colour = (bg_colour[0], bg_colour[1], int(blue+.5))

            #now update the background and caption
            background.fill(bg_colour)
            pygame.display.set_caption("Colour: " + str(bg_colour))

#Here we blit multiple surfaces. Order is important if overlapping.
screen.blit(background, (0, 0))
screen.blit(red_bar, (0, 0))
screen.blit(green_bar, (0, 80)) #Note we have to adjust the top-right corner
screen.blit(blue_bar, (0, 160))
pygame.display.flip()
```

## Surfaces From Images:

All we have seen so far are solid coloured rectangular boxes. While these made a nice simple introduction, they become a little boring quite quickly. Fortunately the exact same Surface object can be used for loading images on to as well b y using the `pygame.image.load()` function:

```
import pygame
from pygame.locals import *
pygame.init()

img = pygame.image.load("lorikeet.bmp") #load an image as a Surface

#use image size to determine the screen size
screen = pygame.display.set_mode(img.get_size())
pygame.display.set_caption("Click pixels to check their colour!")

img = img.convert() #need to convert it after we have set-up the display

#the display will not change, so we can blit & flip the display just once first
screen.blit(img, (0,0))
pygame.display.flip()

clock = pygame.time.Clock()
keep_going = True
while keep_going:
    clock.tick(30)

    for ev in pygame.event.get():
        if ev.type == QUIT:
            keep_going = False
        elif ev.type == MOUSEMOTION:
            pixel = img.get_at(ev.pos)[:3] #just takes (r,g,b), ignores alpha
            pygame.display.set_caption("Pixel Colour: " + str(pixel))
            #Nothing changes, so no need to blit or flip.
```

This example loads an image surface, and uses the size of the image itself to define the display size. It also makes use of the `get_at()` Surface method to determine the colour of specific pixels that the mouse is at when it is moving around. You can also change pixel colours using `set_at()`. Try adding the following to the event handling:

```
elif ev.type == MOUSEBUTTONDOWN:
    img.set_at(ev.pos, (255,255,255)) #sets clicked pixels to white
    screen.blit(img, (0,0)) #Now we need to update & refresh
    pygame.display.flip()
```

You could also access the properties of the mouse directly (like we did with the keyboard earlier) in order to handle the button being held down. However, this will still only check the status every frame, and so moving the mouse quickly will not create a solid line. Try adding *below* the for-loop:

```
if pygame.mouse.get_pressed()[0]: #if left mouse is pressed
    img.set_at(pygame.mouse.get_pos(), (255,255,255))
    screen.blit(img, (0,0))
    pygame.display.flip()
```

## Transparency:

The example above used `get_at()` to get the colour of a pixel. This gives back a colour tuple that has 4 elements. The 4<sup>th</sup> element is called the *alpha channel*, and represents the transparency. It will be an int between 0 (completely transparent) and 255 (completely opaque, the default). Some image types can contain this transparency information (.gif, .png, and even .bmp), and instead of `.convert()` you will use `.convert_alpha()` when loading them. You can also create your own transparency on an entire Surface when it doesn't have any using its `set_alpha()` method. You can also select a single colour to be fully transparent using the Surface's `set_colorkey(colour)` method. This can be useful for non-rectangular images without an alpha channel.

## Animation

So now we can load images as Surfaces, and position those surfaces on the screen when blitting. Animation is a pretty simple addition – we just need to make where we place the image a variable. The best way to manage that position is to use a special pygame type called a Rect. As we will see later Surfaces and Rects are very closely linked. When blitting a Surface, the top-left coordinates can be replaced with a Rect object. Also, a Surface generates its own Rect automatically. A Rect object has several coordinate properties that you can use to control its position. There is a lot of redundancy here, but some are more convenient to use at different times:

- top, left, bottom, right, centerx, centery → *single integers for the borders & center*
- center → *tuple for center of the Surface (perhaps the most useful one!)*
- topleft, bottomleft, topright, bottomright → *(x, y) tuples for corners*
- midtop, midleft, midbottom, midright → *tuples for middles of each border*
- size, width, height → *tuple and single integers for rect size (Note, this doesn't stretch)*

The following example loads an image and moves diagonally by adjusting the left and top of the rectangle.

```
#import & init stuff omitted
screen = pygame.display.set_mode((640, 480))
img = pygame.image.load("lorikeet.bmp").convert()
img_rect = img.get_rect() #the Surface can give us its corresponding Rect
background = pygame.Surface(screen.get_size()).convert()
background.fill((255, 255, 255))
screen.blit(background, (0,0))
screen.blit(img, img_rect)

clock = pygame.time.Clock()
keep_going = True
while keep_going:
    clock.tick(30)
    for ev in pygame.event.get():
        if ev.type == QUIT:
            keep_going = False

    img_rect.top += 1 #changes the Rect's top-left y-coord
    img_rect.left += 1 #changes the Rect's top-left x-coord
    screen.blit(img, img_rect) #blits the image based on the Rect's position
    pygame.display.flip()
```

There are several options for moving the Rect. Instead of changing the top and left, we could have changed centerx and centery, or right and bottom. We could also use the Rect's move() or move\_ip() methods:

```
img_rect = img_rect.move(1,1) #doesn't change the position without reassignment
img_rect.move_ip(1,1) #moves Rect 'in place', reassignment not needed
```

In this simple animation, the image leaves a 'streak' behind it. This is because the image was blitted onto the screen only in the area defined by the Rect. As the Rect moved, the area drawn over also moved, but the previous marks were never erased. The simple fix for this is to first re-blit the entire background to erase the screen, then to blit the image at its new location.

Add the following line before you blit the image: `screen.blit(background, (0,0))`

Movement can be more interesting by handling borders, changing direction and speed. The following ball-bouncing animation is a simple demonstration.

```

#import & init stuff omitted
screen = pygame.display.set_mode((640, 480))
img = pygame.image.load("ball.bmp").convert()
img_rect = img.get_rect()

background = pygame.Surface(screen.get_size()).convert()
background.fill((255, 255, 255))

dir_x = 1 #1 means right, -1 means left
dir_y = 1 #1 means down, -1 means up
speed = 4

clock = pygame.time.Clock()
keep_going = True
while keep_going:
    clock.tick(30)
    for ev in pygame.event.get():
        if ev.type == QUIT:
            keep_going = False
        elif ev.type == KEYDOWN:
            #Ability to increase/decrease speed using up/down arrows
            if ev.key == K_UP:
                speed += 1
            elif ev.key == K_DOWN:
                speed -= 1

    #Handle the walls by changing direction(s)
    if img_rect.left < 0 or img_rect.right >= screen.get_width():
        dir_x *= -1
    if img_rect.top < 0 or img_rect.bottom >= screen.get_height():
        dir_y *= -1

    #update the position
    img_rect.move_ip(speed*dir_x, speed*dir_y)
    #note that this could still move outside the walls slightly

    screen.blit(background, (0,0))
    screen.blit(img, img_rect)
    pygame.display.flip()

```

## Sounds

It would be nice if the ball made a bouncing sound when it hit a wall. You can add sound effects to your games by loading Sound objects, and then telling those Sound objects to play when necessary. Sounds can be .wav, or .ogg .

```

pygame.mixer.pre_init(22050, -16, 2, 1024) #Add before pygame.init()

```

Now where you created your Surfaces you can create a Sound object:

```

bounce = pygame.mixer.Sound("ball_bounce.ogg")
bounce.set_volume(1.0) # 0.0 to 1.0. This is not required!

```

Now, whenever you wish you play the sound (likely in your game loop somewhere):

```

bounce.play() #you can also call repeat using bounce.play(num_repeats)

```

For music files (like .mp3), you will use a slightly different approach (may not work on all systems):

```

pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3) #to adjust the volume
pygame.mixer.music.play(-1) #to play the loaded song, -1 means repeat

```

With this approach, only one piece of music can be loaded at a time (but effects can be played on top).

**\*\*\*Audacity is a great free program that can convert sounds to OGG-Vorbis format, or for creating your own sound effects!**

## Fonts & Text

Output statements are incredibly common in most programming exercises. Even without a full 'game', you may want to have some sort of text feedback to the user. So far we have only been able to do this by changing the window caption.

When you add text to a Surface, you are actually adding small images and manipulating pixels. A piece of text in Pygame is represented as a Surface itself. First you need to set-up a font object, and then you can *render* text Surfaces from it.

To set-up a font object to create text Surfaces from you can use custom fonts or system fonts:

```
my_font = pygame.font.Font("sample.ttf", 60) #custom font, needs font file  
my_font = pygame.font.SysFont("comicsansms", 48) #system fonts, needs font name
```

Each approach needs the name/filename of the font, and the font size. If you use the `Font` approach, you need to have the font file. If you use the `SysFont` approach, you need to know the name of the font on the system, and you need to make sure the user has that same font. If they don't, a default font will be used. You can get a list of system font names using `pygame.font.get_fonts()`.

Regardless of how you set-up the font object, you can create the text Surface using `render()`:

```
label = my_font.render("Pygame!", True, (255,255,0))
```

There are 3 required arguments: the text (a str), a bool for antialiased, and a colour tuple. The remainder of the rectangular Surface created will be transparent by default. Now you can blit the surface wherever you want.

```
#Import & Initialize omitted  
screen = pygame.display.set_mode((400, 200))  
background = pygame.Surface(screen.get_size()).convert()  
background.fill((255, 255, 255))  
  
my_font = pygame.font.Font("sample.ttf", 60)  
my_font2 = pygame.font.SysFont("comicsansms", 48)  
label = my_font.render("Pygame!", True, (255,0,0)) #antialiased  
label2 = my_font2.render("Pygame!", False, (0,0, 255)) #not antialiased  
  
screen.blit(background, (0,0))  
screen.blit(label, (60, 20))  
screen.blit(label2, (60, 80))  
pygame.display.flip()  
#Event loop omitted
```

To further customize the Fonts, there are the methods `set_bold(bool)`, `set_italics(bool)`, and `set_underline(bool)`.

The following is a useful code snippet to that will allow the user to type in text as if it was a text field, and the text on the screen will update. To do so we need to grab the unicode property from the `KEYDOWN` event, append it to the current text string, recreate the text surface, wipe out the old one, and re-blit the new one. While not perfect, this approach makes for a nice exercise. Using the `size()` method would allow you to check the width & height of a Surface required to make the text, which can help with positioning, restricting width, adjusting font size on the fly, etc.

```
#Import & Initialize omitted  
screen = pygame.display.set_mode((400, 100))  
background = pygame.Surface(screen.get_size()).convert()  
background.fill((0, 0, 0))  
  
field_surf = pygame.Surface((300, 50)).convert()  
field_surf.fill((255,255,255))
```

```

my_font = pygame.font.SysFont("helvetica", 20)
field_value = ""
field = my_font.render(field_value, True, (0,0,0))

clock = pygame.time.Clock()
keepGoing = True
while keepGoing:
    clock.tick(30)
    for ev in pygame.event.get():
        if ev.type == QUIT:
            keepGoing = False
        elif ev.type == KEYDOWN:
            if ev.key == K_BACKSPACE and len(field_value) > 0:
                field_value = field_value[:-1] #cut off last character
            elif ev.unicode.isalnum() and len(field_value) < 20:
                #That condition limits the length to 20 and only alphanumeric
                field_value += ev.unicode #adds character value of key
            field = my_font.render(field_value, True, (0,0,0))

    screen.blit(background, (0, 0))
    screen.blit(field_surf, (50, 25))
    screen.blit(field, (60, 40))
    pygame.display.flip()

```

## Sprites

With the ball bouncing example, we maintained two separate properties for the ball: it's Surface, and it's Rect (for positioning). We also maintained 3 variables for horizontal/vertical direction, and speed. What if we had 2, or 3, or 10 of these balls on the screen? We would need to maintain all of this information for each ball separately. This is a perfect justification for making a class to define a ball's properties and behaviour. Game elements like the ball are generally called **sprites**. Every sprite has a visual representation, position and size properties, and should have the *ability* to move (though they may not).

We could easily make our own Python class. However, having a class to model a sprite is so useful that one already exists – `pygame.sprite.Sprite`. This class is not intended to be used on its own, but rather to be customized using *inheritance*. In other words, any game element you will make will become a subclass of the Sprite class:

```

class My_Sprite(pygame.sprite.Sprite):
    ...

```

### Sprite Attributes:

In order for the customized Sprite to work properly, you must create the following instance variables:

- **image** – the Surface for the image that is the sprite, likely loaded using `pygame.image.load()`
- **rect** – a Rect object (more specifically `pygame.rect.Rect`) which models a rectangular area in which the sprite resides. The position/coordinates of this Rect defines the position of the sprite. Generally you will get this Rect from the image itself using `image.get_rect()` so that the rect is the same size as the image (but you can make it smaller or larger if you wish).
- You can also add additional properties, such as our directional and speed properties

### Sprite Methods:

Besides the standard `__init__()` method, there is one other important method that your Sprite will likely implement. Though not required, this method will be useful when using groups later:

- **update()** – you will override this in order to define how the sprite should be updated (e.g. its position). This will likely be called once every frame.
- You can add whatever other methods you want! With respect to the `__init__()`, you can structure it to take as many arguments as you wish.

The following would be a basic but reasonable implementation of a Sprite class for our bouncing ball. Notice that it has essentially all the same elements, but we have wrapped up the Ball properties in a class, and defined its movement behaviour within the class itself instead of within the game loop.

```
class Ball(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self) #construct the parent component
        self.image = pygame.image.load("ball.bmp").convert()
        self.rect = self.image.get_rect() #loads the rect from the image

        #set the position, direction, and speed of the ball
        self.rect.topleft = (10, 10)
        self.dir_x = 1
        self.dir_y = 1
        self.speed = 4

    def update(self):
        #Handle the walls by changing direction(s)
        if self.rect.left < 0 or self.rect.right >= screen.get_width():
            self.dir_x *= -1
        if self.rect.top < 0 or self.rect.bottom >= screen.get_height():
            self.dir_y *= -1
        self.rect.move_ip(self.speed*self.dir_x, self.speed*self.dir_y)

    def adjust_speed(self, delta):
        self.speed += delta

#Program starts here
screen = pygame.display.set_mode((640, 480))
ball = Ball() #Create a Ball sprite

background = pygame.Surface(screen.get_size()).convert()
background.fill((255, 255, 255))

clock = pygame.time.Clock()
keep_going = True
while keep_going:
    clock.tick(30)
    for ev in pygame.event.get():
        if ev.type == QUIT:
            keep_going = False
        elif ev.type == KEYDOWN:
            #Respond to events here, but let Ball class change the the properties
            if ev.key == K_UP:
                ball.adjust_speed(1)
            elif ev.key == K_DOWN:
                ball.adjust_speed(-1)

    ball.update() #Tell the ball to do whatever it needs to do to update
    screen.blit(background, (0,0))
    screen.blit(ball.image, ball.rect) #There will be a better way to do this soon
    pygame.display.flip()
```

## Sprite Groups

Sprites generally work together as a group. Therefore, it would be a lot easier to tell an entire group of Sprites to move all at once as opposed to telling each individual one. There are also built-in conveniences when it comes to collision detection and redrawing them on the screen.

You can create a group using:

```
group_name = pygame.sprite.Group() or
group_name = pygame.sprite.Group(a_sprite) or
group_name = pygame.sprite.Group(a_list_of_sprites)
```

You can add sprites to the group now using: `group_name.add(a_sprite or list_of_sprites)`

### Sprite Group Methods:

Once you have a group of Sprites assembled, you can tell the group as a whole to do something instead of each individual Sprite. The following methods are useful:

- **clear**(destination, Surface) – clears off all the Sprites that were drawn on the destination in the previous frame by replacing with the Surface *in the Sprite's rect area only*. This will commonly be the command: `group_name.clear(screen, background)`. This is *way* more efficient than re-blitting the entire background (most of which remains the same!). Because of this, you likely only need to blit the background surface **once** before the game loop even starts!
- **update**( ) – this will call the `update( )` method of every Sprite object that is in the group (which is why it is useful to have `update( )` defined in your Sprites).
- **draw**(destination) – this will blit each Sprite's image surface on to the destination Surface at the Sprite's rect position. The only downside is that you cannot control the order that the Sprites are stored in the Group, and thus cannot control the order that they are drawn. If order is important, make your group a `pygame.sprite.OrderedUpdates( )` type instead of a `pygame.sprite.Group( )`. Drawing is a little slower as a result, but the order drawn to the screen will match the order that the items are in the Group (or were added originally).
- **sprites**( ) – this will return a list of all the Sprite objects contained within the group.
- **remove**(a\_sprite or list\_of\_sprites) – this will remove a Sprite (or list of them) from the group.

```
screen = pygame.display.set_mode((640, 480))
ball = Ball()
ball_group = pygame.sprite.Group(ball)

background = pygame.Surface(screen.get_size()).convert()
background.fill((255, 255, 255))
screen.blit(background, (0,0)) #Only need to blit this once outside of the loop

clock = pygame.time.Clock()
keep_going = True
while keep_going:
    clock.tick(30)
    for ev in pygame.event.get():
        if ev.type == QUIT:
            keep_going = False
        elif ev.type == KEYDOWN:
            if ev.key == K_UP:
                ball.adjust_speed(1)
            elif ev.key == K_DOWN:
                ball.adjust_speed(-1)
    #These three lines now replace the blitting approach
    ball_group.clear(screen, background)
    ball_group.update()
    ball_group.draw(screen)
pygame.display.flip()
```

## Checking Collisions

In essentially any game, Sprites interact with each other in some way, and you will want to respond accordingly. Did a bullet hit a target? Did the hero run into a barrier? Did the ball hit the paddle? There are several tools in Pygame for exploring collisions.

### Using a Rect:

You can check if one Rect is colliding with a point or another Rect:

- `some_rect.collidect(other_rect)` – returns True if `some_rect` is overlapping/colliding with the `other_rect` in any way (not necessarily entirely), or False otherwise
- `some_rect.collidepoint(point)` – returns True if `some_rect` is overlapping/on a point (an x,y tuple), or False otherwise
- `some_rect.contains(other_rect)` – returns True if `other_rect` is *entirely* contained within `some_rect`, or False otherwise
- `some_rect.collidelist(some_list)` – returns True if any rect in `some_list` is overlapping/colliding with `some_rect` in any way (not necessarily entirely), or False otherwise. The list can be a list of Rects, or a list of objects that have a 'rect' property – like Sprites!
- `some_rect.collidelistall(some_list)` – same as above, but returns a list of indices matching the Rects that did collide with `some_rect`

These methods are significantly simpler than checking the borders of all edges of the Sprites/Rects yourself!

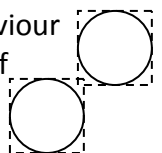
### Using a function from `pygame.sprite`:

The following functions are defined in `pygame.sprite`, and use Sprites and Groups (of any kind) to detect collisions (using the `rect` property of those Sprites):

- `pygame.sprite.spritecollide(sprite, group, dokill)` – returns a list of Sprites in the Group that are colliding with the `sprite` argument. The `dokill` argument is a bool – True if you want the colliding sprites removed from the group, False otherwise
- `pygame.sprite.spritecollideany(sprite, group)` – returns reference to a Sprite that is being collided with, or None. This return type can be treated like a bool: True if the Sprite argument is colliding with any Sprite in the Group, or False otherwise
- `pygame.sprite.collide_rect(sprite1, sprite2)` – returns True if the two Sprite arguments are colliding, or False otherwise (much like `rect.collidect()` above)
- `pygame.sprite.groupcollide(group1, group2, dokill1, dokill2)` – returns a dict of all Sprites in `group1` as keys, and their values are lists of Sprites they are colliding with in `group2`. The `dokill` arguments are bools for removing colliding Sprites from the groups.

Often you will want to check if a Sprite is colliding with any other Sprite in the same Group. Care needs to be taken here because the Sprite will be treated as colliding with itself. One approach is to remove the sprite from the group, then check collisions, then re-add it to the group. Another approach is to get the list of Sprites that are colliding, and check if there are Sprites other than itself.

Also, be aware that since these collisions are all based on the `rect` property, some funny behaviour may occur. If the image used for the sprite has extra space around it, that space is in fact part of the Sprite, and will be part of the collidable area. For the round ball, picture a 'square' wrapping around the circular ball, and notice the strange 'phantom corner' collisions.



You can spend some (or a lot) of time exploring other collision techniques, including other built-in functions, calculating Pythagorean distance from center for circular Sprites, using masks, and pixel-perfect collision.

## Joysticks

There are all sorts of joysticks that can be used as input devices to games. For the most part they are all the same basic set of buttons a different layout. When working on a Windows-based machine, the X-Box controllers work great since the driver is already installed. All you have to do is plug-in the USB connection, and the controller is ready to fire off events to your game. These controllers open up a huge extra dimension for your games, and you will find even basic *'left-right-shoot'* games more enjoyable on a controller than on a keyboard. In addition, the gameplay element can make a simple game completely different through the use of a different controller. For example, imagine PacMan on a keyboard, or on a standard controller. The controls are simple, and gameplay is pretty straight forward. Now imagine changing the controller to a dance-pad, where you control PacMan with your feet. The gameplay changes dramatically, even though your event handling logic will be identical!

### Setup:

You can decide to have a single/specific number of joysticks, or you can ask how many there are plugged in and initialize that many. The following is a simple block of code to initialize all joysticks that are plugged in, and give them appropriate ID numbers (0 → #joysticks-1):

```
#Set-up the joystick(s)
pygame.joystick.init()
for i in range(pygame.joystick.get_count()):
    try:
        j = pygame.joystick.Joystick(i) # create a joystick instance
        j.init() # init instance
        print 'Enabled joystick: ' + j.get_name()
    except pygame.error:
        print 'No joystick #%i found! Please fix this!' %(i)
```

Each joystick will have its own id#, so you can determine which 'player' caused the event.

### Events:

Earlier you saw a table of event types with their properties. This code will show you the significance of each event and the events values. Plug in a joystick and test it out!

```
# Joystick Event Handling
for ev in pygame.event.get():
    if ev.type == pygame.locals.JOYAXISMOTION: #a LOT of these!
        print "Axis #%i on joystick id#%i -->%s" %(ev.axis, ev.joy, ev.value)
    elif ev.type == pygame.locals.JOYHATMOTION: #The up/down/left/right
        print "Presssed hat#%i on joystick id#%i -->%s" %(ev.hat, ev.joy, ev.value)
        #The value tuples are (x,y), with -1 meaning left/down, 1 meaning right/up
    elif ev.type == pygame.locals.JOYBUTTONDOWN: #any button pressed
        print "Pressed button #%i on joystick id#%i" %(ev.button, ev.joy)
    elif ev.type == pygame.locals.JOYBUTTONUP: #any button released
        print "Released button #%i on joystick id#%i" %(ev.button, ev.joy)
```

You do not need to respond to all of these event types. You will notice that there are a lot of the 'axis' events. Each axis is a 2-directional axis, and will have a float value between -1 and 1. The 'analog sticks' that can be rolled around are actually 2 separate axes. You can combine their values with a little basic trigonometry to determine the angled direction of the stick, or you can convert the values into relative speeds, or you can simply look at their left/right status (e.g. for moving a 'Pong' paddle). Because these axes are rather sensitive (often just picking up the controller will fire off events), you may wish to ignore very small values.

Like with the keyboard and mouse, you can also look at the state of the joystick(s) directly without worrying about the events. Each Joystick object has methods that can tell you how many buttons/hats/axes it has, and then using their numbers can tell you the status of each button/hat/axis.

## **Further Topics**

These notes provide you with plenty to chew on, but you may want to also explore the following:

- Transformations – the `pygame.transform` module allows you to rotate, scale, flip, etc. a Surface. Care needs to be taken because as Surfaces are transformed, their rects also change size.
- Grid-based games – Setting up a game to be constrained to a grid can make for some simplified logic. Games like Tron, Snake, Pacman, Zelda, and even Mario can work nicely with a grid structure.
- Vector physics – Some basic physics can make your gameplay much more realistic. Students taking Physics in grade 12 should have the concepts they need if they wish to explore it. There are plenty of good resources.
- Pixel perfect collisions – there are algorithms and even source code available for this ‘perfect’ approach to collisions. However, the processing requirements increase dramatically, so using `rect’s` is still the best way, if possible.
- DirtyRect and DirtySprite – These provide some improved performance by limiting updates to Sprites that have actually changed (or become ‘dirty’) in the frame.
- Animated Sprites – Instead of a Sprite having a single image Surface, you can store a list of them and have the sprite cycle through them over time. Creating the images is the difficult part, but there are many good resources for sprite sheets and ‘tilesets’ of free image collections to use for these sprites.
- 3D – This gets significantly more complex, but you could explore PyOpenGL. There are also some simpler ways to toy around with 3D and Python using PyOgre or Blender.

## **Resources**

There are an increasing number of free Pygame resources, including several on the `pygame.org` website. Below is a list of just a few you can check out!

- Book: Wiley’s “*Game Programming: The Express Line to Learning*” by Andy Harris
- eBook: “*Invent With Python*” by Al Sweigart (Free online!)
- Book: “*Game Programming With Python*” by Sean Riley
- Book: “*Beginning Game Development with Python and Pygame: From Novice to Professional*” by Will McGugan
- Book: “*Hello World! Computer Programming for Kids and Other Beginners*” by Warren & Carter Sande
- Website: <http://thepythongamebook.com> (free *growing* Wiki with lots of code!)
- Resource: [http://gamemaking.indiangames.net/index\\_files/FreeSpritesforGames.htm](http://gamemaking.indiangames.net/index_files/FreeSpritesforGames.htm) (links to many useful resource sites)
- 
- 
- 
- 
-

## **Ideas & Notes**

Use the space below to record any ideas for how Pygame can be used in ICS2/3/4, interesting games exercises, notes, or further areas to explore.