Pygame - Game Notes - Shootemup

- Dr Nick Hayward

A brief intro to developing a *shootemup* game with Pygame.

Contents

    - show player's health - status bar
- Fun game extras
    - update health status colours
    - repetitive firing sequence
    - explosions for sprite objects
- References

Intro

This game will a basic example of a shoot-em-up style shooter, popularised by games such as *Space Invaders*.

Game variables

We can set some initial game variables to help recreate the initial look of an arcade screen for *Space Invaders* &c. e.g.

```
winWidth = 480
winHeight = 640
FPS = 60
```

We'll define the game window as vertical to match an arcade cabinet style screen. We'll also set an initially higher framerate of 60FPS, which is appropriate for this style of action game.

Player object

We can then add a sprite for our first player, a ship to shoot and destroy the advancing aliens.

We'll start by creating a Player class to help abstract many of the required attributes. e.g.

```
# create a default player sprite for the game
class Player(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.Surface((30, 30))
        self.image.fill(WHITE)
        self.rect = self.image.get_rect() #checks images and get rect...
        self.rect.centerx = winWidth / 2
        self.rect.bottom = winHeight - 20
        self.speed_x = 0

    # update per loop iteration
    def update(self):
        self.rect.x += self.speed_x
```

So, we can create a sprite for our player, add an initial rectangle for testing, and then fill it with a white colour. Then, we need to specify a bounding rectangle for the sprite, and set a start position in the game window. We can use built-in functions to get the relative position of our sprite's rectangle, and then set the coordinates on the game window.

Then, we can add our initial update function, which will run each iteration of the active loop. So, for the first example we're simply setting the speed of updates to the x-axis for our sprite.

show in game window

After creating the sprite, we'll need to setup our sprite group, and specific player sprite. e.g.

```
# game sprite group
game_sprites = pygame.sprite.Group()
# create player object
player = Player()
# add sprite to game's sprite group
game_sprites.add(player)
```

Add events

We need to add listeners to the event part of our game loop. For example, we may add an initial listener for the exit option for the game window itself.

For this type of game, we'll start by setting a few defaults for the motion of the sprite. These will also influence and effect how we use listeners for events that will control the player.

So, we can now add a default speed and a basic keypress listener to the update function in our Player class. e.g.

```
...
def update(self):
    self.speed_x = 0
    key_state = pygame.key.get_pressed()
    if key_state[pygame.K_LEFT]:
        self.speed_x = -5
    if key_state[pygame.K_RIGHT]:
        self.speed_x = 5
    self.rect.x += self.speed_x
...
```

We set the object's speed to 0 to ensure that it is only ever moving when a player requests movement for the sprite object. We can then setup a listener for a keypress on the right and left directional arrows.

So, in this example, when a player presses either the left or right directional arrow, a listener will be able to update the speed of the sprite to enable horizontal movement along the x-axis for the set speed.

We can then update this code to handle the sprite object as it reaches the horizontal edges of the game window. e.g.

```
def update(self):
    ...
    if self.rect.right > winWidth:
        self.rect.right = winWidth
    if self.rect.left < 0:
        self.rect.left = 0
```

We simply add boundary checks for the sprite to the update function in the Player class. As it reaches either side we can check the bounding for either the max width of the game window or the min width. Then, we reset the position of the rect for the sprite's object.

Add enemy objects

We can now start to add our game's enemy objects, which are commonly given a collective, generic name of *mob*. We can add the following class Mob to our game,

```
# create a generic enemy sprite for the game - standard name is *mob*
class Mob(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.Surface((20, 20))
        self.image.fill(CYAN)
        # specify bounding rect for sprite
        self.rect = self.image.get_rect()
        # specify random start posn & speed of enemies
        self.rect.x = random.randrange(winWidth - self.rect.width)
        self.rect.y = random.randrange(-100, -50)
        self.speed_y = random.randrange(1, 10)

    def update(self):
        self.rect.y += self.speed_y
```

With this class, we can create an enemy sprite, set its size, colour, &c. and then set random x and y coordinates for the starting position of the enemy. We use random values to ensure that the enemies start and move from different positions at the top of the game window, and progress in staggered groups down the window.

update enemy objects

As our enemy objects move down the game window, we need to check if and when they leave the bottom of the game window. We can add the following checks to the update function,

```
    def update(self):
        self.rect.y += self.speed_y
        # check if enemy sprite leaves the bottom of the game window - then
randomise at the top...
        if self.rect.top > winHeight + 15:
            # specify random start posn & speed of enemies
            self.rect.x = random.randrange(winWidth - self.rect.width)
            self.rect.y = random.randrange(-100, -50)
            self.speed_y = random.randrange(1, 7)
```

So, as the enemy sprite leaves the bottom of the game window, we can check its position. Then, we may reset the enemy sprite object to the top of the game window. However, we need to ensure that the same sprite object does not simply loop around, and reappear at the same position at the top of the game window. This would become both too easy and tedious for our player.

So, we simply reset our *mob* object to a random path down the window. Hopefully, this will make it slightly harder for our player. We can also ensure that each enemy sprite has a different speed by simply randomising the speed along the y-axis per sprite object.

show enemy objects

We can now create a *mob* group as a container for our enemy objects. This group will become particularly useful as we add collision detection later in the game. So, we can update our code as follows, e.g.

```
# sprite groups - game, mob...
mob_sprites = pygame.sprite.Group()
# create enemy objects, add to sprite groups...
for i in range(10):
    mob = Mob()
    # add to game_sprites group to get object updated
    game_sprites.add(mob)
    # add to mob_sprites group - use for collision detection &c.
    mob_sprites.add(mob)
```

We create our *mob* objects, and then simply add them to the required sprite groups. By adding them to the game_sprites group, they will be updated as the game loop is executed. The mob_sprites group will help us easily detect these enemy objects when we need to add collision detection, remove them from the game window, and so on.

modify motion of enemy objects

Whilst the above updates work great for random motion along the y-axis, we can add some variation to the movement of an enemy object by simply modifying the x-axis. For example, we can modify the x-axis for each enemy object to create variant angular motion along both the x-axis and y-axis.

```
# random speed along the x-axis
self.speed_x = random.randrange(-3, 3)
...

self.rect.x += self.speed_x
# check if enemy sprite leaves the bottom of the game window - then randomise at the
top...
if self.rect.top > winHeight + 15 or self.rect.left < -15 or self.rect.right >
winWidth + 15:
    # specify random start posn & speed of enemies
    self.rect.x = random.randrange(winWidth - self.rect.width)
    self.speed_x = random.randrange(-3, 3)
...
```

So, our mob class may now be updated as follows,

```
# create a generic enemy sprite for the game - standard name is *mob*
class Mob(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.Surface((20, 20))
        self.image.fill(CYAN)
        # specify bounding rect for sprite
        self.rect = self.image.get_rect()
        # specify random start posn & speed of enemies
        self.rect.x = random.randrange(winWidth - self.rect.width)
        self.rect.y = random.randrange(-100, -50)
        # random speed along the x-axis
        self.speed_x = random.randrange(-3, 3)
        # random speed along the y-axis
        self.speed_y = random.randrange(1, 7)

    def update(self):
        self.rect.x += self.speed_x
        self.rect.y += self.speed_y
        # check if enemy sprite leaves the bottom of the game window - then
randomise at the top...
        if self.rect.top > winHeight + 15 or self.rect.left < -15 or self.rect.right
```

```
> winWidth + 15:
            # specify random start posn & speed of enemies
            self.rect.x = random.randrange(winWidth - self.rect.width)
            self.rect.y = random.randrange(-100, -50)
            self.speed_x = random.randrange(-3, 3)
            self.speed_y = random.randrange(1, 7)
```

We've also added a quick check for the motion of our enemy sprite object along the x-axis. So, as the sprite exits on either side of the screen, we can simply create a new sprite on a random path down the screen.

Collision detection

Pygame includes support for adding explicit collision detection between two or more sprites in a game window. We can use built-in functions to help us work with these collisions.

basic collisions

We need to add basic collision detection for each time an enemy object hits the player's object at the foot of the game window. To help with this detection, Pygame includes the following function,

```
# add check for collision - enemy and player sprites (False = hit object is not
deleted from game window)
pygame.sprite.spritecollide(player, mob_sprites, False)
```

This function of the sprite object allows us to check if one sprite object has been hit by another sprite object. In this example, we're checking to see if the player sprite object has been hit by an enemy sprite object from the mob_sprites group. The False parameter is a boolean value for the state of the object that has hit. i.e. this determines whether an enemy sprite object should be deleted from the game window or not.

This command is particularly useful as it returns a *list* data structure containing any of the enemy sprite objects that hit the player sprite object. So, we may now update this code as follows, and store this list in a variable

```
hits = pygame.sprite.spritecollide(player, mob_sprites, False)
```

We may then use this *list* to check if any collisions have occurred in our game window, e.g.

```
...
if collisions:
    # update game objects &c.
    ...
...
```

We can use the above code to simply return a boolean value to check if the *list* collisions is empty or not.

Add shooting objects

To create some projectiles for shooting, such as bullets or laser beams, we can create a new class for this sprite object. e.g.

```
# create a generic projectile sprite - for bullets, lasers &c.
class Projectile(pygame.sprite.Sprite):
    # x, y - add specific location for object relative to player sprite
    def __init__(self, x, y):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.Surface((5, 10))
        self.image.fill(RED)
        self.rect = self.image.get_rect()
        # weapon fired from front (top) of player sprite...
        self.rect.bottom = y
        self.rect.centerx = x
        # speed of projectile up the screen
        self.speed_y = -10

    def update(self):
        # update y relative to speed of projectile on y-axis
        self.rect.y += self.speed_y
        # remove from game window - if it goes beyond bounding for y-axis at top...
        if self.rect.bottom < 0:
            # kill() removes specified sprite from group...
            self.kill()
```

In this class, we're creating another sprite object for a projectile such as a bullet or a laser beam. This projectile will be shot from the top of the player object, so we may set the x and y coordinates relative to the position of this player object. We're setting the speed along the y-axis so it travels up the screen.

As we update each projectile object, we simply update its speed, and then check its position on the screen. If it leaves the top of the game window, we can call the generic kill() method on this sprite. This method is available for any sprite object we create in the game window.

shoot the projectile

Then, we need to add a new listener to the game loop to detect a keypress for the *spacebar*. We'll use this keypress to allow a player to shoot these projectiles, a laser beam for example.

```
# 'processing' inputs (events)
for event in EVENTS.get():
    # check keyboard events - keydown
    if event.type == pygame.KEYDOWN:
        # check for ESCAPE key
        if event.key == pygame.K_ESCAPE:
            gameExit()
        elif event.key == pygame.K_SPACE:
            # fire laser beam...
            player.fire()
```

In this example, we've updated our keypress listeners to include a check for each time a player hits down on the *spacebar*. We can use this keypress event to fire our projectile, a laser beam to hit our enemy mobs.

So, we now need to update our Player class to include a method for firing the projectiles from the top of the player's sprite object. For example,

```
# fire projectile from top of player sprite object
def fire(self):
    # set position of projectile relative to player's object rect for centerx and
top
    projectile = Projectile(self.rect.centerx, self.rect.top)
    # add projectile to game sprites group
    game_sprites.add(projectile)
    # add each projectile to sprite group for all projectiles
    projectiles.add(projectile)
```

This sets the start position for the x and y coordinates of each projectile sprite to the current position of the player's sprite object. Then, we simply add each projectile sprite object to the main game sprite group, and a new sprite group for all of the projectiles. We can add this new sprite group as follows,

```
projectiles = pygame.sprite.Group()
```

Now, when a player presses down on the *spacebar* a projectile, our red laser beam, will be fired from the top of the player's sprite object.

Destroy enemy objects

We can now add collision detection for the projectile, our laser beams, hitting an enemy object. In effect, we have one group of sprites that may be colliding with another, defined sprite group. So, we may use Pygame's collide method for sprite groups, e.g.

```
# add check for sprite group collide with another sprite group - projectiles hitting
enemy objects - use True to delete sprites from each group...
collisions = pygame.sprite.groupcollide(mob_sprites, projectiles, True, True)
```

The boolean parameter values of True and True allow us to delete both the hit enemy objects, and the projectile objects that hit them.

Then, as the *list* of collisions is populated, we may create new enemy objects for those that have been hit and deleted.

```
# add more mobs for those hit and deleted by projectiles
for collision in collisions:
    mob = Mob()
    game_sprites.add(mob)
    mob_sprites.add(mob)
```

If we don't create new enemy objects, the game window will quickly run out of enemy sprite objects.

Not much fun for the player!

Add graphics

We can now add images and backgrounds to our shooter game to help represent the player's ship, laser beams firing, asteroids to hit, and star-filled background.

Before we can add our images for the sprites and backgrounds, we need to add some image files to our game's directory structure. For example, we'll normally create an assets folder, and then add any required images, audio, video &c. to sub-directories for use within our game.

So, we may now update our directory structure to include the required assets,

```
|-- shootemup
    |-- assets
        |-- images
            |__ ship.png
```

and so on.

import assets

In our Pygame code, we'll need to import the Python module for os, which will then allow us to query a local OS's directory structure.

```
# import os
import os
```

Then, we can specify the directory location of the main game file, so Python can keep track of the relative location of this file. For example,

```
game_dir = os.path.dirname(__file__)
```

__file__ is used by Python to abstract the root application file, which is then portable from system to system.

This allows us to set relative paths for game directories, for example

```
# game assets
game_dir = os.path.dirname(__file__)
# relative path to assets dir
assets_dir = os.path.join(game_dir, "assets")
# relative path to image dir
img_dir = os.path.join(assets_dir, "images")
```

We may then import an image for use as a sprite as follows,

```
# assets - images
ship = pygame.image.load(os.path.join(img_dir, "ship.png"))
```

convert

As we import an image for use as a sprite within our game, we need to use a convert() method to ensure that the image file is of a type Pygame can use natively. If not, there is a potential for the game to perform more slowly. For example,

```
ship = pygame.image.load(os.path.join(img_dir, "ship.png")).convert()
```

In effect, Pygame uses this method, convert(), to create a copy of the image. This image copy is then drawn a lot faster to the Pygame window.

colour key

For each image that Pygame adds as a sprite, a bounding rectangle will be set with a given colour.

However, in most examples, we want to set the background of our sprite to transparent. This means the rectangle for the image will now blend with the background colour of our game window. For example,

```
ball.set_colorkey(WHITE)
```

This will now check for white coloured pixels in the image background, and then set them to transparent.

add game background

We can now add a background image for our game. For example, we may wish to recreate stars and space. For example, we can add our background as follows,

```
# load graphics
bg_img = pygame.image.load(os.path.join(img_dir, "bg-purple.png")).convert
```

We can also add a rectangle to contain our background image,

```
# add rect for bg - helps locate background
bg_rect = bg_img.get_rect()
```

This basically helps us know where to add our background image, and then subsequently find it as needed with the logic of our game.

We can then draw our background image as part of the game loop,

```
# draw background image - specify image file and rect to load image
window.blit(bg_img, bg_rect)
```

add game images

Then, we need to add an image for our player's ship, laser beams, and asteroids to shoot at.

We can add these as follows

```
# add ship image
ship_img = pygame.image.load(os.path.join(img_dir, "ship-blue.png")).convert()
# ship's laser
laser_img = pygame.image.load(os.path.join(img_dir, "laser-blue.png")).convert()
# asteroid
asteroid_img = pygame.image.load(os.path.join(img_dir, "asteroid-med-
grey.png")).convert()
```

To use these new images in our game, we then need to modify the code for each object. For example, for our Player object, we can update our class to include a reference to the ship_img

```
self.image = ship_img
```

However, we can also customise this image by scaling it to better fit our game window,

```
# load ship image & scale to fit game window...
self.image = pygame.transform.scale(ship_img, (49, 37))
# set colorkey to remove black background for ship's rect
self.image.set_colorkey(BLACK)
```

We can also update our ship's rect using a colorkey to ensure the black rect is not visible in the game window.

Add better collision detection

Our collision detection is currently using rectangles to detect one sprite colliding with another. This is technically referred to as follows,

- Axis-aligned Bounding Box (AABB)

However, for some sprite images this will often cause an unrealistic effect as the two images collide. In effect, the image does not appear to collide with the other image due to space caused by each respective rectangle. So, as one corner of a rectangle hits another corner a collision will be detected.

```
 ________
|        |
|        |
|       _|_____
|______|_|     |
       |       |
       |_______|
```

As this example shows, unless each sprite's image fits exactly inside the respective bounding box, there will be space left over.

Now, we have a few options for rectifying this issue. We might choose to simply calculate and set a slightly smaller rectangle as the required bounding box. Another option, for example, might simply be to use a circular bounding box for our sprite image.

The benefit of using a rectangle, an *axis-aligned bounding box*, is that the game is able to detect and calculate collisions much faster for a rectangle bounding box.

However, a *circular bounding box* may be slower. This is simply due to the number of calculations the game may need to perform to check radius of one bounding box against another bounding box radius. Thankfully, this rarely becomes a practical issue unless you're trying to work with thousands of potential sprite images.

One other option, and the most precise, is to use *pixel perfect collision detection*. As the name might suggest, the game engine will check each pixel of each possible sprite image to determine if and when they collided. It's particularly resource intensive unless you require such precision.

add circle bounding box

We can add some *circle bounding boxes* to our sprite images, in particular for the player's ship and mob objects. We can start by adding explicit circles with a fill colour, which help us check the relative position of the circle's bounding box,

```
self.radius = 20
pygame.draw.circle(self.image, RED, self.rect.center, self.radius)
```

As we know the sprite image for the player's ship will have a fixed, known size we may set the radius to 20, for example.

Then, we may add some *circle bounding boxes* to the mob objects as well,

```
self.radius = int(self.rect.width * 0.9 / 2)
pygame.draw.circle(self.image, RED, self.rect.center, self.radius)
```

We've used the same basic pattern to add circles, but for the mob objects we may set each circle's radius relative to the sprite image. So, we're setting the radius as 90% of the width of the sprite image, and then returning half of that value.

To be able to use each *circle bounding box*, we need to update the collision checks as well. We can simply update this check for each mob object the *update* section of the game loop as follows,

```
# add check for collision - enemy and player sprites (False = hit object is not deleted from game window)
collisions = pygame.sprite.spritecollide(player, mob_sprites, False, pygame.sprite.collide_circle)
```

We've updated the collision check to explicitly look for circle collisions. With this update, we can now remove the explicit drawn circle for each *circle bounding box* for the player and mob object sprites. For example, we may simply comment out the drawn circle

```
self.radius = int(self.rect.width * 0.9 / 2)
#pygame.draw.circle(self.image, RED, self.rect.center, self.radius)
```

Animating sprite images

For many sprites, it's fun and useful to be able to animate different states, actions, and so on as the game progresses. For example, we might consider animating the destruction of a mob sprite object as it's destroyed by a laser from the player's ship. Or, perhaps we might make our game slightly more interesting for our player if these mob sprite objects were moving on the screen.

So, let's consider how we may now add animations to our mob sprite objects. We've already added scale transform to these objects, and we may use the same pattern to add a rotate option to add some semblance of animation to these sprites as they move down the game window.

For example, we may start by setting some variables for our rotation,

```
# set up rotation for sprite image - default rotate value, rotate speed to add diff.
directions,
self.rotate = 0
self.rotate_speed = random.randrange(-7, 7)
```

However, due to the framerate of this game, set to 60FPS, we need to ensure that the rotate animation does not occur for each update of the game loop. If it did, the rotation would be too quick, not realistic, and run the risk of annoying our player.

Therefore, in addition to the rotate animation, we also need to consider how to create a timer for this animation. In effect, the regularity of the update to the animation to ensure it renders realistically. There is already a timer available within our existing code, which we're currently using to monitor the framerate for our game. However, we may also use this timer to check the last time we updated our mob sprite image. We can set a time to rotate the sprite image, and then check this monitor as it reaches this specified time.

So, we may record the last time our sprite image was rotated by getting the time, the number of ticks, since the game started. This value will be recorded as follows,

```
# check timer for last update to rotate
self.rotate_update = pygame.time.get_ticks()
```

Each time the mob sprite image object is rotated, we can update the value of this variable to record the last time for a rotation.

We can modify the mob sprite's update function as follows,

```
# call rotate update
self.rotate()
```

where we're simply going to call a separate rotate function. This helps to keep the code cleaner and easier to read, and also allows us to quickly and easily modify, remove, and simply stop our object's rotation.

So, we can now add our new rotate() function, and start by checking if it's time to rotate the sprite image

```
def rotate(self):
        # check time - get time now and check if ready to rotate sprite image
        time_now = pygame.time.get_ticks()
        # check if ready to update...in milliseconds
        if time_now - self.rotate_update > 70:
            self.last_update = time_now
```

This simply uses the current time, relative to the game's timer, and then checks this value against the last value for a rotate update. If the difference is greater than 70 milliseconds, it's time to rotate the sprite object.

rotate issues

For the rotation itself, we can't simply add a *rotate transform* to the rotate() function. Whilst this is possible in the code, it will also cause the game window to practically freeze, thereby making the game unplayable. For example,

```
self.image = pygame.transform.rotate(self.image, self.rotate_speed)
```

The reason is that each rotation of a sprite object image causes the game logic to lose part of the pixels for that image. If we continuously rotate an image, the game will not be able to keep up with the pixels for the image. So, the game loop then starts to freeze.

correct rotation

We may correct the above issue by working with an original, pristine image for the sprite object.

```
# set pristine original image for sprite object
self.image_original = asteroid_img
# set colour key for original image
self.image_original.set_colorkey(BLACK)
```

Then, we can set the initial sprite object image as a copy of this original,

```
# set copy image for sprite rendering
self.image = self.image_original.copy()
```

We can then use the pristine original image with the rotation,

```
self.image = pygame.transform.rotate(self.image_original, self.rotate_speed)
```

correct rotation speed

Another issue we need to fix is the rotation speed for a sprite object image. If we simply use our default self.rotate_speed, we're not keeping track of how far we've actually rotated the image. So, we need to keep a record of incremental rotation of the image to ensure that it rotates smoothly and in a realistic manner.

We can monitor this rotation by using the value of the rotation, and then adding the rotation speed for each update to a sprite object image. Then, as the image rotates we can simply check its value as a modulus of 360 to ensure it keeps rotating correctly.

```
self.rotate = (self.rotate + self.rotate_speed) % 360
self.image = pygame.transform.rotate(self.image_original, self.rotate)
```

rect rotation issues

However, we still have an issue with the *rectangle* bounding box, which does not rotate. As the sprite image rotates, it loses its centre relative to the bounding rectangle.

To correct this issue, we can now modify our logic for the sprite object's *update* as follows,

```
# new image for rotation
rotate_image = pygame.transform.rotate(self.image_original, self.rotate)
# check location of original centre of rect
original_centre = self.rect.center
# set image to rotate image
self.image = rotate_image
# create new rect for image
self.rect = self.image.get_rect()
self.rect.center = original_centre
```

Our mob sprite object images will now correctly rotate as they move down the screen.

Random mob sprite images

To add random image, at least randomised from potential options, we need to add a list of available images for the random selection,

```
asteroid_list = ["asteroid-tiny-grey.png", "asteroid-small-grey.png", "asteroid-med-grey.png"]
```

We also need a new list for our asteroid images,

```
asteroid_imgs = []
```

Then, we simply need to loop through this asteroid list, and then add each available image to the list of asteroid_images.

For example,

```
for img in asteroid_list:
    asteroid_imgs.append(pygame.image.load(os.path.join(img_dir, img)).convert())
```

We may then update the Mob class to set a random image from the asteroid_imgs list, e.g.

```
self.image_original = random.choice(asteroid_imgs)
```

The images for our mob sprite objects will now be randomly chosen from the available list of images.

Keep a player's game score

We can now start to consider how to record a score for our player in this example game. We'll start by adding an initial variable to record the player's score in the game, e.g.

```
# initialise game score - default to 0
game_score = 0
```

Then, we need to allow our player to score points for each projectile collision on a mob object, i.e. when a laser beam hits an asteroid. It might also be fun to set variant points relative to the size of the mob object. Again, depending upon how we specify these scores, we can either pre-define them or use the relative sizes to calculate the points per mob object.

For example, if we use the radius of each mob object, we may then perform a quick calculation for each collision to work out points per asteroid,

```
# calculate points relative to size of mob object
game_score += 40 - collision.radius
```

So, relative to the recorded collision, we can simply get the radius per hit mob object, and then minus from a known starting value. The number of points will then be set relative to the size of each mob object.

Render text

Drawing text to a game window in Pygame can become a repetitive process as part of each window update. So, we may abstract this underlying game requirement to a text output function,

```
 # text output and render function - draw to game window
def textRender(surface, text, size, x, y):
    # specify font for text render
    ...
```

We start by specifying a surface where we need to draw the text, then the text to render, its size, and the coordinates relative to the specified surface. We also need to specify a font for the text to be rendered. As with web development, we're reliant upon the installed fonts for the user's local system. However, we may use a font-match function with Pygame, which helps abstract the specification of an exact font to a relative name. For example,

```
# specify font name to find
font_match = pygame.font.match_font('arial')
```

Pygame will then search the local system for a font with the specified name. We can, of course, also include a custom font file with each game, but this simple option helps abstract the font process for each game.

We may now use this specified font to create an object for the font we need to render text in the game window, e.g.

```
# specify font for text render - uses found font and size of text
font = pygame.font.Font(font_match, size)
```

The text we'll be adding to the game window needs to be drawn, effectively pixel by pixel. So, Pygame calculates the drawing for each pixel to create the specified text in the required font. We can start by specifying a surface to draw the required pixels for the text, e.g.

```
# surface for text pixels - TRUE = anti-aliased
text_surface = font.render(text, True, WHITE)
```

In this example, we're specifying where to draw the text, the text to draw to the game window, a boolean value for *anti-aliasing* of the text, and the text colour.

Then, we need to calculate a rectangle for placing the text surface, e.g.

```
# get rect for text surface rendering
text_rect = text_surface.get_rect()
```

and, we can then specify where to position our text surface relative to the defined x and y coordinates, e.g.

```
# specify a relative location for text
text_rect.midtop = (x, y)
```

This text is then added to the surface using the standard blit function, e.g.

```
# add text surface to location of text rect
surface.blit(text_surface, text_rect)
```

So, the created text_surface, which contains the rendered text, is itself added to the location of the text_rect on the overall specified surface. In most examples, this overall surface will simply be the main game window surface.

Our overall text draw function is now as follows,

```
# text output and render function - draw to game window
def textRender(surface, text, size, x, y):
    # specify font for text render - uses found font and size of text
    font = pygame.font.Font(font_match, size)
    # surface for text pixels - TRUE = anti-aliased
    text_surface = font.render(text, True, WHITE)
    # get rect for text surface rendering
    text_rect = text_surface.get_rect()
    # specify a relative location for text
    text_rect.midtop = (x, y)
    # add text surface to location of text rect
    surface.blit(text_surface, text_rect)
```

We may now call this function whenever we need to render text to our game window.

render text to game window

We may now add some text to our game window to test that the textRender() is working correctly.

In the draw section of our game loop, we may now add the following call, e.g.

```
# draw text to game window - game score
textRender(window, str(game_score), 18, winWidth / 2, 10)
```

This will call the textRender() function, specifying the surface as the overall game window, the string to render, our score for example, the font size, and then the x and y coordinates for rendering the text.

We now have a game, which enables our player to move their ship, shoot and destroy asteroids, and record a score as the game progresses.

Game music and sound effects

For a game's sound effects, there are many different options and sources for these sounds.

We may try open source examples, such as

- Open Game Art

or perhaps create our own custom sounds using a utility such as **SFXR**, or its derivative online option at the following website,

- BFXR

For most of these sound effects, we'll be using a *WAV* format for the sound files. We may also use other file formats such as *OGG*.

We can then add these files for our sound effects to the game assets directory,

```
|-- shootemup
    |-- assets
        |-- images
            |__ ship.png
        |-- sounds
            |__ laser-beam-med.wav
            |__ explosion-med.wav
```

import sounds and effects

The first thing we need to add is support for Pygame's mixer. For example, we may add the following call after we initialise Pygame itself,

```
# add sound mixer to game
pygame.mixer.init()
```

To use these sounds and effects in our game window, we need to add the directory location, as we did for images. For example,

```
# relative path to music and sound effects dir
snd_dir = os.path.join(assets_dir, "sounds")
```

We can then start to add our required music and sound effects, again following a similar pattern to loading and using images. So, we can add our sound files as follows,

```
# load music and sound effects for use in game window
# laser beam firing sound effect
laser_effect = pygame.mixer.Sound(os.path.join(snd_dir, 'laser-beam-med.wav'))
# explosion sound effect
explosion_effect = pygame.mixer.Sound(os.path.join(snd_dir, 'explosion-med.wav'))
```

We can add these lines of code right after we've loaded our images, just before we start the game loop itself.

use sound effects

After importing and loading our sound effects, we may then choose where we need to play these sound effects.

For example, as a player fires the laser beam to destroy falling mob objects, we may also call the appropriate sound effect. e.g.

```
# fire projectile from top of player sprite object
def fire(self):
    ...
    # play laser beam sound effect
    laser_effect.play()
```

So, each time a player now fires at the projectiles an accompanying sound effect will be played.

We can also add sound effects for each mob object explosion. We'll add the following play call as part of the check for collisions,

```
# play laser beam sound effect
laser_effect.play()
```

We now have a sound effect for firing our ship's laser beam, and as a mob object, our asteroids, explode when hit.

use music in a game

As we add sound effects, we may also load music to play in the game.

For example, we may want music playing in the background whilst a player enjoys the game. So, we can load our required background music as follows,

```
# load music for background playback in game window
pygame.mixer.music.load(os.path.join(snd_dir, 'space-music-bg.ogg'))
```

We may also use a standard *WAV* file for the music, which helps compatibility across multiple systems.

n.b. some MP3 files have notable issues playing back correctly on OS X, which is a shame. *WAV* and *OGG* have been tested OK.

We can also set a relative volume for this background music, in effect creating ambience and not overwhelming the player. e.g.

```
# set music volume - half standard volume
pygame.mixer.music.set_volume(0.5)
```

Check player's health

Our current game only gives a player one chance to shoot and destroy any advancing mob objects before a single hit ends the game.

A single hit may make the game more challenging, but it is certainly not what most players would expect for this type of game. So, we may now consider monitoring and updating the status of a player's health as they are hit by advancing mob objects, i.e. our falling and rotating asteroids.

One way to protect our player, and their ship, is to use a *Star Trek* style shield. This shield will offer full protection initially, and then incrementally weaken with each hit from a mob object until it eventually fails at value 0. We'll set a default for this player shield in the Player class,

```
# set default health for our player - start at max 100% and then decrease...
self.stShield = 100
```

Then, we need to modify our logic for a mob collision to ensure that the way we handle such objects better reflects a decrease in the player's shield, and health.

For example, instead of allowing a mob object to continue after it has collided with the player, we now need to remove it from the game window. If we don't update this boolean to True, each mob object will simply continue to hit the player as it moves, pixel by pixel, through the player's ship. A single hit would quickly become compounded in the gameplay.

```
# add check for collision - enemy and player sprites (True = hit object is now
deleted from game window)
collisions = pygame.sprite.spritecollide(player, mob_sprites, True,
pygame.sprite.collide_circle)
```

Then, as our player may be hit by multiple mob objects, we also need to update our check from a simple conditional to a loop through the possible collisions,

```
# check collisions with player's ship - decrease shield for each hit
for collision in collisions:
    # decrease player's shield for each collision
    player.stShield -= 20
    # check overall shield value - quit game if no shield
    if player.stShield <= 0:
        running = False
```

So, our player's shield will now survive four direct collisions before the fifth will destroy the ship, and then end the game.

replace mob objects

Whilst this now works as expected, we have an issue with losing mob objects if they collide with the player's ship. This follows the same underlying pattern as the player's laser beam firing on the asteroids, our mob objects.

So, we need to create a new object if it is removed after a collision. As this is a familiar pattern, we may now abstract the creation of the mob objects to avoid repetition of code, e.g.

```
# create a mob object
def createMob():
    mob = Mob()
    # add to game_sprites group to get object updated
    game_sprites.add(mob)
    # add to mob_sprites group - use for collision detection &c.
    mob_sprites.add(mob)
```

This simple abstracted function now allows us to easily recreate our mob objects by creating a mob object, adding it to the overall group of game_sprites, and then the specific group for the game's mob_sprites.

We can then call this abstracted function whenever a mob object collides with a projectile, or the player's ship. We may also call this function when we initially create our new mob objects as part of the loop to 10.

```
# create a new mob object
createMob()
```

show player's health - status bar

We've already defined a default maximum for our player's shield, and we can now start to output its value to the game window.

Whilst we could simply output a numerical value, as we did for the player's score, it seems more interesting to show a graphical update for the status of a player's health.

So, we can define a new draw function to allow us to render a visual health bar for the player's shield,

```
# draw a status bar for the player's health - percentage of health
def drawStatusBar(surface, x, y, health_pct):
    # defaults for status bar dimension
    BAR_WIDTH = 100
    BAR_HEIGHT = 10
    # use health as percentage to calculate fill for status bar
    bar_fill = (health / 100) * BAR_WIDTH
    # rectangles - outline of status bar &
    bar_rect = pygame.Rect(x, y, BAR_WIDTH, BAR_HEIGHT)
    fill_rect = pygame.Rect(x, y, bar_fill, BAR_HEIGHT)
    # draw health status bar to the game window - 2 specifies pixels for border
width
    pygame.draw.rect(surface, GREEN, fill_rect)
    pygame.draw.rect(surface, WHITE, bar_rect, 2)
```

This function accepts four parameters, which allow us to easily define a surface for rendering, its `x` and `y` location in the game window, and then update the status of the player's health. In this case, we're simply updating the health of the shield for the player's ship.

We can set a default width and height for the status bar, and then specify how much of this bar needs to be filled with colour relative to the player's current health status. This health status can be calculated as a percentage, which then allows us to easily modify the relative sizes for the status bar.

We may also specify our rectangles for the status bar, which includes an outer container for the status bar, and effectively an inner bar for the colour fill to represent the player's health.

Our current health status bar will now start by showing 100% fill, and then reduce by 20% for each collision between a mob object, one of our asteroids, and the player's ship. When the health hits 0, the game will then quit.

Fun game extras

We can now start to add some fun extras to the general gameplay to help improve the player experience. For example, we might consider modifying our health status bar to better reflect health percentages, auto fire for the laser beam so our player may hold down the space bar to continuously shoot, and fun explosions for collisions.

update health status colours

So, we may start such fun extras by slightly modifying the health status bar to more accurately inform the player of the health of their ship. A common option is to simply modify the colour of the status bar to reflect this health status. So, we may use a bright colour to indicate greater health status, and then change it to RED as a warning to the player. e.g.

```
if bar_fill < 40:
    pygame.draw.rect(surface, RED, fill_rect)
else:
    pygame.draw.rect(surface, CYAN, fill_rect)
```

So, we're just adding a conditional check for the percentage value for the player's health, and then modifying the colour of the fill for the health status bar.

repetitive firing sequence

In our current game logic, as a user presses down on the space bar, a laser beam will be fired from the top of the player's ship. However, one press is equal to one firing sequence.

So, we need to still check that the spacebar has been pressed down, but now continue to fire a laser beam until the key is released.

In our Player class, we can add a couple of new variables. The first is to specify the delay in milliseconds between each firing of the laser beam, and the second allows us to check the time, the number of ticks, since the last beam was fired, e.g.

```
...
# firing delay between laser beams
self.firing_delay = 200
# time in ms since last fired
self.last_fired = pygame.time.get_ticks()
```

We may then add a listener for the space bar event to the update() method in the Player class.

```
# check space bar for firing projectile
if key_state[pygame.K_SPACE]:
    # fire laser beam
    self.fire()
```

Then, we can update our fire() method to reflect this repetitive firing sequence, e.g.

```
...
# get current time
time_now = pygame.time.get_ticks()
if time_now - self.last_fired > self.firing_delay:
    self.last_fired = time_now
    ...
```

So, our fire() method has now been updated as follows,

```
# fire projectile from top of player sprite object
def fire(self):
    # get current time
    time_now = pygame.time.get_ticks()
    if time_now - self.last_fired > self.firing_delay:
        self.last_fired = time_now
        # set position of projectile relative to player's object rect for centerx
and top
        projectile = Projectile(self.rect.centerx, self.rect.top)
        # add projectile to game sprites group
        game_sprites.add(projectile)
        # add each projectile to sprite group for all projectiles
        projectiles.add(projectile)
        # play laser beam sound effect
        laser_effect.play()
```

We can now also remove the listener for a spacebar event in the events section of the game loop.

explosions for sprite objects

To create realistic explosion effects for our sprite objects, we need to create or import a series of images for these explosions.

We may then use these defined images to create an animated sequence for each required explosion. In effect, as we've seen with standard animation, we specify a starting image and then cycle or animate our way through this set of images.

In our game, we may introduce this animation for each projectile collision with an asteroid, or perhaps as our player's ship finally runs out of lives and explodes.

These are fun extras that add a sense of achievement to the underlying challange of destroying advancing asteroids or alien ships.

load explosion images

As we've done so far for other game sprite objects, we need to be able to define and load our images for the explosions. We'll use a list for these images, as we did for the random asteroids, and then we can cycle through these explosions as required.

So, our first example will use a list to simply load these explosion images. As we know the directory for these images, and the required number of images, we'll use a for loop to iterate over this directory and load our images, e.g.

```
# explosions
explosion_imgs = []

# iterate over explosion images in directory
for i in range(9):
    file = 'explosion{}.png'.format(i)
    expl_img = pygame.image.load(os.path.join(img_dir, file)).convert()
    expl_img.set_colorkey(BLACK)
    explosion_imgs.append(expl_img)
```

As we loop through the directory of images, we can use the built-in function, format(), to specify an abstracted value for the iterator number, in this example, of the created filename. So, for each iteration, the value for the {} braces will be replaced with the index value of the iterator.

Then, we may create our image for the Pygame window, and set the colour key to black to create our transparency for the containing shape's background.

We can then append these images to our list for explosions.

create explosion sprite object

As we've done for other sprite objects, we may now create a new class to help us represent and organise our sprite object for *explosions*.

So, we'll add a new class for this object, and then start by initialising this sprite, e.g.

```
# create a generic explosion sprite - use for asteroids, player explosions &c.
class Explosion(pygame.sprite.Sprite):
    # initialise sprite
    def __init__(self, center):
        pygame.sprite.Sprite.__init__(self)
        ...
```

After initialising this new sprite object, we can set the starting image for our explosions to the first index position of our list for the explosion images. Then, we need to add the rectangle for this image, and set its centre to the specified value of the passed parameter.

We also need to set the initial frame for our animation, which we can set to a starting default of 0.

As we're working with an animation, which is cycling through images, we need to try to make this steady and constant. One way to achieve this desired animation effect is to create a steady framerate for the animation itself. So, we may now check the time in ticks for the last update, following the pattern we've seen earlier in our game.

Then, we can set a default framerate for this animation. We can test this animation, and modify this framerate as required.

```
# create a generic explosion sprite - use for asteroids, player explosions &c.
class Explosion(pygame.sprite.Sprite):
    # initialise sprite
    def __init__(self, center):
        pygame.sprite.Sprite.__init__(self)
        # specify image for explosion sprite
        self.image = explosion_imgs[0]
        # set rect for image
        self.rect = self.image.get_rect()
        self.rect.center = center
        # set initial frame for animation
        self.frame = 0
        # check last update to animation
        self.last_update = pygame.time.get_ticks()
        # set framerate delay between animation frames - sets speed for explosion
        self.frame_rate = 50
```

Then, we need to add an update function to our class, which will allow us to update the image of the explosion for this sprite object as time progresses. i.e. as the framerate advances, we can switch the explosion images to create the animation itself.

```
...
# change image as time progresses for explosion sprite
def update(self):
    # get current time
    now = pygame.time.get_ticks()
    # check if enough time has passed between animations
    if now - self.last_update > self.frame_rate:
        self.last_update = now
        # if enough time passed - add 1 to frame
        self.frame += 1
        # check if end of explosion images reached
        if self.frame == len(explosion_imgs):
            # kill if end of image reached
            self.kill()
        else:
            center = self.rect.center
            self.image = explosion_imgs[self.frame]
            # update rect for image
            self.rect = self.image.get_rect()
            self.rect.center = center
```

In this update function, we need to check the current time in the game, which then allows us to check if enough time has passed between each animation. If enough time has elapsed, we can update the value for the last_update time record, and advance our animation frame by an increment of 1. i.e. we can move to the next available image in the series of explosions.

If we reach the end of these explosion images, i.e. when enough frames have passed, we can then end or kill() this animation for the explosions. If not, we can set the centre of our explosion image's rectangle, and update the image itself.

add explosions to collisions

We've now created our sprite object for explosions, but we still need to use this object in the logic of our game.

So, we can now call this explosion whenever we record a collision between, for example, a projectile and an asteroid.

In the game loop's update section, as we check for collisions we can now add an animation for the explosions. e.g.

```
...
# add more mobs for those hit and deleted by projectiles
for collision in collisions:
    # calculate points relative to size of mob object
    game_score += 40 - collision.radius
    # play explosion sound effect for collision
    explosion_effect.play()
    # add animation for explosion images if collision
    explosion = Explosion(collision.rect.center)
    # add explosion sprite to game sprites group
    game_sprites.add(explosion)
    # create a new mob object
    createMob()
...
```

As we're checking for collisions, we can now create the animation for the explosions after we play the sound effect for each explosion.

add explosions to player's ship

Adding these explosions to another sprite object, such as collisions against the player's ship, is as simple as updating the game loop again. So, as we decrease the relative level for the shield of our player's ship, we can then trigger an explosion to reinforce this collision for our player. e.g.

```
# add check for collision - enemy and player sprites (True = hit object is now
deleted from game window)
collisions = pygame.sprite.spritecollide(player, mob_sprites, True,
pygame.sprite.collide_circle)
# check collisions with player's ship - decrease shield for each hit
for collision in collisions:
    # decrease player's shield for each collision
    player.stShield -= 20
    # add animation for explosion images if collision
    explosion = Explosion(collision.rect.center)
    # add explosion sprite to game sprites group
    game_sprites.add(explosion)
    # create a new mob object
    createMob()
    # check overall shield value - quit game if no shield
    if player.stShield <= 0:
        running = False
```

scale explosion images

Our explosions are now being shown for both initial types of collision within our game. As a projectile hits a mob object, and then as a cascading mob object strikes the shield of our player's ship.

However, there is still a lingering issue with these explosions that is not reinforcing the gameplay for our shooter style game. Effectively, there is no differentiation in the relative size of an explosion, and therefore no semblance of feedback to our player.

So, we might add a standard scale transform to the image for each explosion sprite object,

```
# explosions
explosion_imgs = []

# iterate over explosion images in directory
for i in range(9):
    file = 'explosion{}.png'.format(i)
    # load image from os
    expl_img = pygame.image.load(os.path.join(img_dir, file)).convert()
    # set colour key for image
    expl_img.set_colorkey(BLACK)
    # append to specified list for explosion images
    explosion_imgs.append(expl_img)
```

This gives us a smaller, less overwhelming explosion for each mob object, and collision against the player's ship.

However, it would also be useful to be able to scale these explosions relative to the actual size of a given sprite object. So, a smaller relative explosion image for a smaller mob object, and likewise for a collision against the player's ship.

The first thing we need to do is update our class for the Explosion object, which will allow us to dynamically modify each explosion image in the animation relative to a specified size. In effect, we can scale each frame of the explosion animation to match the size of the collision object.

So, we can update this class as follows,

```
# create a generic explosion sprite - use for asteroids, player explosions &c.
class Explosion(pygame.sprite.Sprite):
    # initialise sprite
    def __init__(self, center, size):
        pygame.sprite.Sprite.__init__(self)
        # specify size for explosion sprite
        self.size = size
        # get initial image for explosion
        self.image = pygame.transform.scale(explosion_imgs[0], self.size)
...
```

We'll start by adding a parameter for size, which allows us to pass a variable size for each collision object. We can then use this size to scale the initial image for the explosion animation.

As we update this object, each frame of the animation will also require scaling of the explosion image. e.g.

```
# change image as time progresses for explosion sprite
def update(self):
    # get current time
    now = pygame.time.get_ticks()
    # check if enough time has passed between animations
    if now - self.last_update > self.frame_rate:
        self.last_update = now
        # if enough time passed - add 1 to frame
        self.frame += 1
        # check if end of explosion images reached
        if self.frame == len(explosion_imgs):
            # kill if end of image reached
            self.kill()
        else:
            center = self.rect.center
            self.image = pygame.transform.scale(explosion_imgs[self.frame],
self.size)
            # update rect for image
            self.rect = self.image.get_rect()
            self.rect.center = center
```

The key update is in the block of code for the if/else conditional statement. As we output each frame of the explosion animation, we may then scale this image to match the passed size for the explosion object.

So, different size mob objects will have a matching explosion animation, which we may update in the game loop, e.g.

```
# add check for sprite group collide with another sprite group - projectiles hitting
enemy objects - use True to delete sprites from each group...
collisions = pygame.sprite.groupcollide(mob_sprites, projectiles, True, True)
# add more mobs for those hit and deleted by projectiles
for collision in collisions:
    # calculate points relative to size of mob object
    game_score += 40 - collision.radius
    # play explosion sound effect for collision
    explosion_effect.play()
    # get size of collision object
    col_size = collision.rect.size
    #print("collision size = " + str(col_size))
    # add animation for explosion images if collision
    explosion = Explosion(collision.rect.center, col_size)
    # add explosion sprite to game sprites group
    game_sprites.add(explosion)
    # create a new mob object
    createMob()
```

And, the same for the player's sprite object as well.

References

- Python
    - Python API - os
- Pygame
    - pygame.draw
    - pygame.font
    - pygame.image
    - pygame.key
    - pygame.mixer
    - pygame.sprite
    - pygame.time
    - pygame.transform
- Various
    - BFXR
    - Open Game Art