

# Python

## PyGame for Game Development: Sprite Groups and Collision Detection

Contributed by Peyton McCullough

2006-01-31

In this second part of a tutorial covering PyGame, you will learn how to manage entire groups of sprites. You'll also learn how to test for collisions within your applications.

### Sprite Groups

While managing sprites individually may be ideal for small applications, think about managing dozens of sprites in more complex applications. Obviously, there has to be a way to efficiently manage all of them, and sprite groups do just that. The definition of a sprite group is pretty simple and true to the term. Sprite groups are simply groups of sprites that are related in some way. Sprites can also belong to multiple groups, rather than just being limited to one group. Besides providing a convenient way to iterate over related sprites, some sprite groups contain special features to aid with development, which we'll take a look at later. First, though, let's put together a simple program that uses a sprite group to move three stick men up and down the screen.



We'll start with the most basic sprite group, the *Group* class:

```
import pygame
import sys

class StickMan(pygame.sprite.Sprite):

    # We'll just accept the x-position here
    def __init__(self, xPosition):

        pygame.sprite.Sprite.__init__(self)
        self.old = (0, 0, 0, 0)
        self.image = pygame.image.load('stickMan.gif')
        self.rect = self.image.get_rect()
        self.rect.x = xPosition

    # The x-position remains the same
    def update(self, yPosition):

        self.old = self.rect
        self.rect = self.rect.move([0, yPosition - self.rect.y])
```

```
# Define a function to erase old sprite positions
# This will be used later
def eraseSprite(screen, rect):
    screen.blit(blank, rect)

pygame.init()
screen = pygame.display.set_mode((256, 256))
pygame.display.set_caption('Sprite Groups')
screen.fill((255, 255, 255))

# Create the three stick men
stick1 = StickMan(25)
stick2 = StickMan(75)
stick3 = StickMan(125)

# Create a group and add the sprites
stickGroup = pygame.sprite.Group()
stickGroup.add((stick1, stick2, stick3))

# Add a variable for the direction, y-position and height of the
# sprite we are dealing with
stickGroup.direction = "up"
stickGroup.y = screen.get_rect().centery
stickGroup.height = stick1.rect.height

# Create a blank piece of background
blank = pygame.Surface((stick1.rect.width, stick1.rect.height))
blank.fill((255, 255, 255))

pygame.display.update()

# Create an event that will appear ever 100 milliseconds
# This will be used to update the screen
pygame.time.set_timer(pygame.USEREVENT + 1, 100)

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

        # Check for our update event
        if event.type == pygame.USEREVENT + 1:
            # Update the y-position
            if stickGroup.direction == "up":
                stickGroup.y = stickGroup.y - 10
            else:
                stickGroup.y = stickGroup.y + 10
```

```

    # Check if we have gone off the screen
    # If we have, fix it
    if stickGroup.direction == "up" and stickGroup.y <= 0:
        stickGroup.direction = "down"
    elif stickGroup.direction == "down" and stickGroup.y >=
(screen.get_rect().height - stickGroup.height):
        stickGroup.direction = "up"
        stickGroup.y = screen.get_rect().height -
stickGroup.height

    # Clear the old sprites
    # Notice that we pass our eraseSprite function
    # This will be called, and the screen and old position
will be passed
    stickGroup.clear(screen, eraseSprite)

    # Update the sprites
    stickGroup.update(stickGroup.y)

    # Blit the sprites
    stickGroup.draw(screen)

    # Create a list to store the updated rectangles
    updateRects = []

    # Get the updated rectangles
    for man in stickGroup:
        updateRects.append(man.old)
        updateRects.append(man.rect)
    pygame.display.update(updateRects)

```

We start off by creating our *StickMan* sprite class with `__init__` and update methods. Then, we define a function that will be responsible for clearing the previous position of sprites. This will be used later on in the code by the sprite group. Next, we create three sprites and then a sprite group named *stickGroup* from the *Group* class. We add our sprites to this group and then define some special attributes that hold the direction the sprites will be moving, the y-position of the sprites and the height of the sprites. Since the sprites will be moving in unison, we can do this. We then create an event that will be added to the event queue every one hundred milliseconds. This will be used to trigger screen updates. In our loop, if the event is found in the event queue, we update the y-position of the sprites (actually, the *stickGroup*'s *y* attribute, which is later passed to the sprites), making adjustments and changing the direction if needed.

Now we begin to see the magic of sprite groups. In a single method call, we clear the sprites. We pass *screen* and our *eraseSprite* function we defined earlier. Two arguments, the surface to be drawn on and the area that needs to be cleared, are passed to the *eraseSprite* function. We then update the sprites in another method call, passing the new y-position. This simply calls the *update* method of each of the sprites. Finally, we blit the sprites and update the changed rectangles.

Of course, the benefit doesn't end there. Notice how the section of our script that gets the updated rectangles is multiple lines long. This is easily fixed by using the *RenderUpdates* sprite group class rather than the *Group* class.

```

...
stickGroup = pygame.sprite.RenderUpdates()
...

```

The *RenderUpdates* class returns a list of updated rectangles when we call its draw method. We can then pass these to the update method:

```
...
updateRects = stickGroup.draw(screen)
pygame.display.update(updateRects)
...
```

We can also get rid of *StickMan's old* attribute, since it is no longer needed. *RenderUpdates* finds the old positions automatically and throws them into the list of updated rectangles.

PyGame also contains a class called *GroupSingle* that can contain only a single sprite. When a new sprite is added, the old one removed. Normally, a sprite is removed from a group using the *remove* method. You can also empty the sprite group entirely:

```
>>> import pygame
>>>
>>> spritel = pygame.sprite.Sprite()
>>> sprite2 = pygame.sprite.Sprite()
>>>
>>> group = pygame.sprite.Group()
>>> group.add(spritel, sprite2)
>>>
>>> group.remove(spritel)
>>> print group.sprites()
[<Sprite sprite(in 1 groups)>]
>>>
>>> group.empty()
>>> print group.sprites()
[]
```

If none of PyGame's sprite group classes suit you, it's always possible to subclass an existing class and tweak it to fit your needs. This way, you can modify existing methods and create new ones that are more specific to your application.

Sprite groups also provide a simple way to test for collision within your applications. PyGame contains three methods that allow us to test for collision, each behaving a bit differently than the rest. They are *pygame.sprite.spritecollide*, *pygame.sprite.groupcollide* and *pygame.sprite.collideany*. Let's create an example application featuring three non-playable *StickMan* sprites and one character *StickMan* sprite. A collision between the player and a non-playable sprite will create different results for each sprite:

```
import pygame
import sys

class StickMan(pygame.sprite.Sprite):

    def __init__(self, position):

        pygame.sprite.Sprite.__init__(self)
        self.screen = pygame.display.get_surface().get_rect()
        self.image = pygame.image.load('stickMan.gif')
        self.rect = self.image.get_rect()
```

```
self.rect.x = position[0]
self.rect.y = position[1]

def update(self, amount):

    self.rect = self.rect.move(amount)

    # Check for offscreen movements
    if self.rect.x < 0:
        self.rect.x = 0
    elif self.rect.x > (self.screen.width - self.rect.width):
        self.rect.x = self.screen.width - self.rect.width
    if self.rect.y < 0:
        self.rect.y = 0
    elif self.rect.y > (self.screen.height - self.rect.height):
        self.rect.y = self.screen.height - self.rect.height

# Function to erase the sprite
def eraseSprite(screen, rect):
    screen.blit(blank, rect)

pygame.init()
screen = pygame.display.set_mode((256, 256))
pygame.display.set_caption('Collision Detection')
screen.fill((255, 255, 255))

# Create three non-playable stickmen
stick1 = StickMan((25, 25))
stick2 = StickMan((100, 150))
stick3 = StickMan((175, 175))

# Add them each to a group
group1 = pygame.sprite.Group()
group2 = pygame.sprite.Group()
group3 = pygame.sprite.Group()
group1.add(stick1)
group2.add(stick2)
group3.add(stick3)

# Create a playable stickman and add it to a group
player = StickMan((200,10))
playerGroup = pygame.sprite.RenderUpdates()
playerGroup.add(player)

# Create a background piece
blank = pygame.Surface((player.rect.width, player.rect.height))
blank.fill((255, 255, 255))

# Draw each sprite
group1.draw(screen)
group2.draw(screen)
```

```
group3.draw(screen)
playerGroup.draw(screen)

pygame.display.update()

while True:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()

        # Move the player if a key is pressed
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                player.update([-10, 0])
            elif event.key == pygame.K_UP:
                player.update([0, -10])
            elif event.key == pygame.K_RIGHT:
                player.update([10, 0])
            elif event.key == pygame.K_DOWN:
                player.update([0, 10])

        # Check to see if the player has collided with stick1 in
group1
        # If a collision is detected, kill stick1 (True)
        collide1 = pygame.sprite.spritecollide(player, group1,
True)

        # Check to see if the playerGroup has collided with
group2
        # Kill all involved sprites if there is a collision
        collide2 = pygame.sprite.groupcollide(playerGroup,
group2, True, True)

        # Check to see if the player has collided with anything
in group3
        collide3 = pygame.sprite.spritecollideany(player,
group3)

        # Print the test results
        print 'Test 1:', collide1
        print 'Test 2:', collide2
        print 'Test 3:', collide3

        # If the player has died, print a game over message,
wait and then quit
        if not player.alive():
            print 'Game Over'
            pygame.time.wait(3000)
            sys.exit()
```

```

# Clear the player's old spot
playerGroup.clear(screen, eraseSprite)

# Draw the player
updateRects = playerGroup.draw(screen)

# If we need to redraw other things, then do so
if collide1:
    group1.clear(screen, eraseSprite)
    group1.draw(screen)
    updateRects.append(stick1.rect)
elif collide2:
    group2.draw(screen)
    updateRects.append(stick2.rect)
elif collide3:
    group3.draw(screen)
    updateRects.append(stick3.rect)

# Update everything
pygame.display.update(updateRects)

```

It seems like a lot of code, but most of it should be familiar to you. We start out by creating the *StickMan* sprite class and the sprite erasing function. We then create three non-playable and non-moving sprites and add them each to a group, followed by the player's character and its own group. After we draw each sprite, we enter the game loop. Here, we check to see whether the user has pushed a key. If he or she has, we update the player sprite accordingly.

With the updated positions, we then move on to the collision tests. As I mentioned before, there are three methods that are responsible for this. The first is *spritecollide*, and it accepts three arguments. The first is a sprite, and the second is a group. If the two collide and the third argument is set to *True*, then the colliding sprites are removed from the group. The method returns a list of the colliding sprites. The second method is *groupcollide*, and it accepts two groups and two boolean values. If the first boolean value is set to *True*, then the colliding sprites in the first group are removed. If the second is set to *True*, then the colliding sprites in the second group are removed. A dictionary is returned with the sprites of the first group as the keys and the colliding sprites as the values. The last method is *spritecollideany*. It simply checks to see whether a sprite collides with a group, and it returns a boolean value. Since it doesn't do anything too special, like removing sprites, then it is the fastest.

Next, we check to see whether the player sprite still belongs to a group and hasn't been removed by a collision method. If it has been removed, then we exit the game. Otherwise, we move the player and update the other sprites if we need to.

## Conclusion

PyGame provides an easy interface to graphical tasks (as well as other tasks) for use in Python-powered games or even applications. Fonts are simple to load and display text in, and it's easy to load images and display them to a *Surface* object—the graphical building block of PyGame applications. Moreover, sprites can be easily created and used to simplify the drawing process and provide a more object-oriented approach to it, and sprite groups provide an efficient way to organize sprites and accomplish tasks such as collision detection. If you are looking to build a simple game in Python or an application with a unique interface, consider using PyGame.