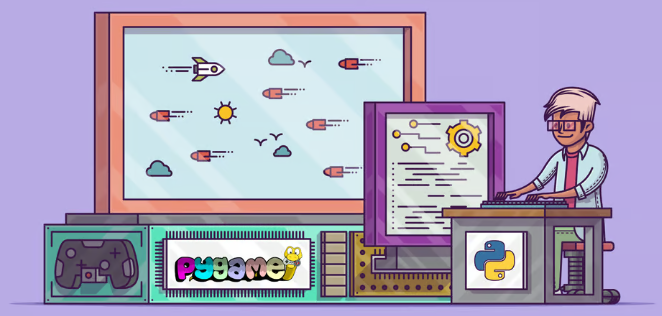
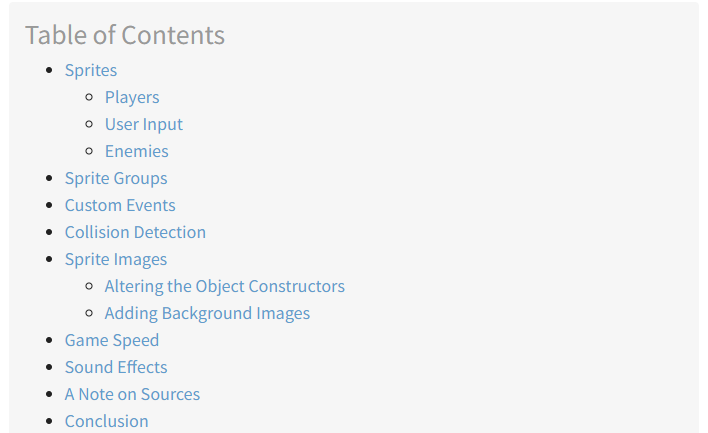
**Session 10**

**Complete Game Development Using PyGame 2**

****



## **Sprites**

In your game design, the player starts on the left, and obstacles come in from the right. You can represent all the obstacles with Surface objects to make drawing everything easier, but how do you know where to draw them? How do you know if an obstacle has collided with the player? What happens when the obstacle flies off the screen? What if you want to draw background images that also move? What if you want your images to be animated? You can handle all these situations and more with [sprites](https://en.wikipedia.org/wiki/Sprite_%28computer_graphics%29).

In programming terms, a **sprite** is a 2D representation of something on the screen. Essentially, it’s a picture. pygame provides a [Sprite class](https://www.pygame.org/docs/ref/sprite.html), which is designed to hold one or several graphical representations of any game object that you want to display on the screen. To use it, you create a new class that extends Sprite. This allows you to use its built-in methods.

### Players

### 

Here’s how you use Sprite objects with the current game to define the player.

Insert this code after line 18:

20# Define a Player object by extending pygame.sprite.Sprite

21# The surface drawn on the screen is now an attribute of 'player'

22class Player(pygame.sprite.Sprite):

23 def \_\_init\_\_(self):

24 super(Player, self).\_\_init\_\_()

25 self.surf = pygame.Surface((75, 25))

26 self.surf.fill((255, 255, 255))

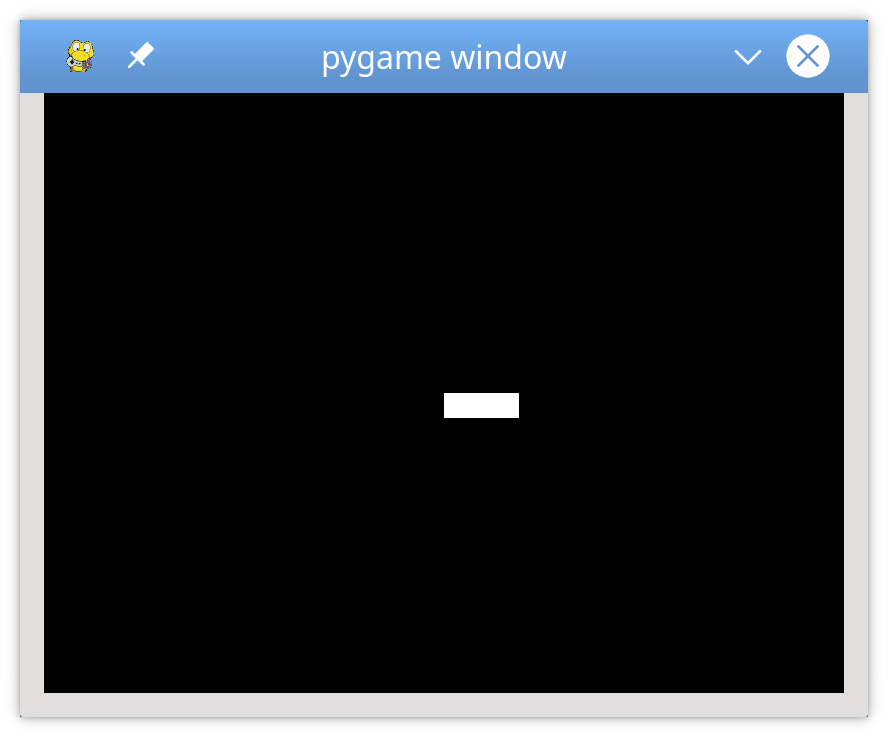
27 self.rect = self.surf.get\_rect()

You first define Player by extending pygame. sprite.Sprite on line 22. Then .\_\_init\_\_() uses .super() to call the .\_\_init\_\_() method of Sprite. For more info on why this is necessary, you can read [Supercharge Your Classes With Python super()](https://realpython.com/python-super/).

Next, you define and initialize .surf to hold the image to display, which is currently a white box. You also define and initialize .rect, which you’ll use to draw the player later. To use this new class, you need to create a new object and change the drawing code as well. Expand the code block below to see it all together:

[Expand for full codes Show/Hide](https://realpython.com/pygame-a-primer/#sprites)

Run this code. You’ll see a white rectangle at roughly the middle of the screen :



What do you think would happen if you changed line 59 to screen. blit(player. surf, player. rect)?

55# Fill the screen with black

56screen.fill((0, 0, 0))

57

58# Draw the player on the screen

59screen.blit(player.surf, player.rect)

60

61# Update the display

62pygame.display.flip()

When you pass a Rect to .blit(), it uses the coordinates of the top left corner to draw the surface. You’ll use this later to make your player move!

### User Input

So far, you’ve learned how to set up pygame and draw objects on the screen. Now, the real fun starts! You’ll make the player controllable using the keyboard.

Earlier, you saw that pygame. event.get() returns a list of the events in the event queue, which you scan for KEYDOWN event types. Well, that’s not the only way to read keypresses. pygame also provides pygame.event.get\_pressed(), which returns a [dictionary](https://realpython.com/python-dicts/) containing all the current KEYDOWN events in the queue.

Put this in your game loop right after the event handling loop. This returns a dictionary containing the keys pressed at the beginning of every frame:

54# Get the set of keys pressed and check for user input

55pressed\_keys = pygame. key.get\_pressed()

Next, you write a method in Player to accept that dictionary. This will define the behavior of the sprite based on the keys that are pressed. Here’s what that might look like:

29# Move the sprite based on user keypresses

30def update(self, pressed\_keys):

31 if pressed\_keys[K\_UP]:

32 self.rect.move\_ip(0, -5)

33 if pressed\_keys[K\_DOWN]:

34 self.rect.move\_ip(0, 5)

35 if pressed\_keys[K\_LEFT]:

36 self.rect.move\_ip(-5, 0)

37 if pressed\_keys[K\_RIGHT]:

38 self. rect.move\_ip(5, 0)

K\_UP, K\_DOWN, K\_LEFT, and K\_RIGHT correspond to the arrow keys on the keyboard. If the dictionary entry for that key is True, then that key is down, and you move the player .rect in the proper direction. Here you use .move\_ip(), which stands for [**move in place**](https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move_ip), to move the current Rect.

Then you can call .update() every frame to move the player sprite in response to keypresses. Add this call right after the call to .get\_pressed():

52# Main loop

53while running:

54 # for loop through the event queue

55 for event in pygame. event.get():

56 # Check for KEYDOWN event

57 if event.type == KEYDOWN:

58 # If the Esc key is pressed, then exit the main loop

59 if event.key == K\_ESCAPE:

60 running = False

61 # Check for QUIT event. If QUIT, then set running to false.

62 elif event.type == QUIT:

63 running = False

64

65 # Get all the keys currently pressed

66 pressed\_keys = pygame. key.get\_pressed()

67

68 # Update the player sprite based on user keypresses

69 player.update(pressed\_keys)

70

71 # Fill the screen with black

72 screen.fill((0, 0, 0))

Now you can move your player rectangle around the screen with the arrow keys:



You may notice two small problems:

1. The player's rectangle can move very fast if a key is held down. You’ll work on that later.
2. The player's rectangle can move off the screen. Let’s solve that one now.

To keep the player on the screen, you need to add some logic to detect if the rect is going to move off-screen. To do that, you check whether the rect coordinates have moved beyond the screen’s boundary. If so, then you instruct the program to move it back to the edge:

25# Move the sprite based on user keypresses

26def update(self, pressed\_keys):

27 if pressed\_keys[K\_UP]:

28 self.rect.move\_ip(0, -5)

29 if pressed\_keys[K\_DOWN]:

30 self.rect.move\_ip(0, 5)

31 if pressed\_keys[K\_LEFT]:

32 self.rect.move\_ip(-5, 0)

33 if pressed\_keys[K\_RIGHT]:

34 self. rect.move\_ip(5, 0)

35

36 # Keep player on the screen

37 if self. rect.left < 0:

38 self. rect.left = 0

39 if self. rect.right > SCREEN\_WIDTH:

40 self. rect.right = SCREEN\_WIDTH

41 if self. rect.top <= 0:

42 self. rect.top = 0

43 if self. rect.bottom >= SCREEN\_HEIGHT:

44 self. rect.bottom = SCREEN\_HEIGHT

Here, instead of using .move(), you just change the corresponding coordinates of .top, .bottom, .left, or .right directly. Test this, and you’ll find the player rectangle can no longer move off the screen.

Now let’s add some enemies!

### Enemies

What’s a game without enemies? You’ll use the techniques you’ve already learned to create a basic enemy class, then create a lot of them for your player to avoid. First, import the random library:

4# Import random for random numbers

5import random

Then create a new sprite class called Enemy, following the same pattern you used for Player:

55# Define the enemy object by extending pygame. sprite.Sprite

56# The surface you draw on the screen is now an attribute of 'enemy'

57class Enemy(pygame.sprite.Sprite):

58 def \_\_init\_\_(self):

59 super(Enemy, self).\_\_init\_\_()

60 self.surf = pygame.Surface((20, 10))

61 self. surf.fill((255, 255, 255))

62 self.rect = self. surf.get\_rect(

63 center=(

64 random.randint(SCREEN\_WIDTH + 20, SCREEN\_WIDTH + 100),

65 random.randint(0, SCREEN\_HEIGHT),

66 )

67 )

68 self.speed = random.randint(5, 20)

69

70 # Move the sprite based on speed

71 # Remove the sprite when it passes the left edge of the screen

72 def update(self):

73 self.rect.move\_ip(-self.speed, 0)

74 if self. rect.right < 0:

75 self.kill()

There are four notable differences between Enemy and Player:

1. **On lines 62 to 67**, you update rect to be a random location along the right edge of the screen. The center of the rectangle is just off the screen. It’s located at some position between 20 and 100 pixels away from the right edge, and somewhere between the top and bottom edges.
2. **On line 68**, you define .speed as a random number between 5 and 20. This specifies how fast this enemy moves towards the player.
3. **On lines 73 to 76**, you define .update(). It takes no arguments since enemies move automatically. Instead, .update() moves the enemy toward the left side of the screen at the .speed defined when it was created.
4. **On line 74**, you check whether the enemy has moved off-screen. To make sure the Enemy is fully off the screen and won’t just disappear while it’s still visible, you check that the right side of the .rect has gone past the left side of the screen. Once the enemy is off-screen, you call .kill() to prevent it from being processed further.

So, what does .kill() do? To figure this out, you have to know about **Sprite Groups**.

## **Sprite Groups**

Another super useful class that pygame provides is the [Sprite Group](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group). This is an object that holds a group of Sprite objects. So why use it? Can’t you just track your Sprite objects in a list instead? Well, you can, but the advantage of using a Group lies in the methods it exposes. These methods help to detect whether any Enemy has collided with the Player, which makes updates much easier.

Let’s see how to create sprite groups. You’ll create two different Group objects:

1. The first Group will hold every Sprite in the game.
2. The second Group will hold just the Enemy objects.

Here’s what that looks like in the code:

82# Create the 'player'

83player = Player()

84

85# Create groups to hold enemy sprites and all sprites

86# - enemies are used for collision detection and position updates

87# - all\_sprites is used for rendering

88enemies = pygame. sprite.Group()

89all\_sprites = pygame. sprite.Group()

90all\_sprites.add(player)

91

92# Variable to keep the main loop running

93running = True

When you call .kill(), the Sprite is removed from every Group to which it belongs. This removes the references to the Sprite as well, which allows [Python’s garbage collector](https://realpython.com/python-memory-management/#garbage-collection) to reclaim the memory as necessary.

Now that you have an all\_sprites group, you can change how objects are drawn. Instead of calling .blit() on just Player, you can iterate over everything in all\_sprites:

117# Fill the screen with black

118screen.fill((0, 0, 0))

119

120# Draw all sprites

121for entity in all\_sprites:

122 screen.blit(entity.surf, entity.rect)

123

124# Flip everything to the display

125pygame.display.flip()

Now, anything put into all\_sprites will be drawn with every frame, whether it’s an enemy or the player.

There’s just one problem… You don’t have any enemies! You could create a bunch of enemies at the beginning of the game, but the game would quickly become boring when they all left the screen a few seconds later. Instead, let’s explore how to keep a steady supply of enemies coming as the game progresses.

## **Custom Events**

The design calls for enemies to appear at regular intervals. This means that at set intervals, you need to do two things:

1. Create a new Enemy.
2. Add it to all\_sprites and enemies.

You already have code that handles random events. The event loop is designed to look for random events occurring in every frame and deal with them appropriately. Luckily, pygame doesn’t restrict you to using only the event types it has defined. You can define your events to handle as you see fit.

Let’s see how to create a custom event that’s generated every few seconds. You can create a custom event by naming it:

78# Create the screen object

79# The size is determined by the constant SCREEN\_WIDTH and SCREEN\_HEIGHT

80screen = pygame. display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

81

82# Create a custom event for adding a new enemy

83ADDENEMY = pygame.USEREVENT + 1

84pygame.time.set\_timer(ADDENEMY, 250)

85

86# Instantiate player. Right now, this is just a rectangle.

87player = Player()

pygame defines events internally as integers, so you need to define a new event with a unique integer. The last event pygame reserves is called USEREVENT, so defining ADDENEMY = pygame.USEREVENT + 1 on line 83 ensures it’s unique.

Next, you need to insert this new event into the event queue at regular intervals throughout the game. That’s where the [time](https://www.pygame.org/docs/ref/time.html) module comes in. Line 84 fires the new ADDENEMY event every 250 milliseconds or four times per second. You call .set\_timer() outside the game loop since you only need one timer, but it will fire throughout the entire game.

Add the code to handle your new event:

100# Main loop

101while running:

102 # Look at every event in the queue

103 for event in pygame. event.get():

104 # Did the user hit a key?

105 if event.type == KEYDOWN:

106 # Was it the Escape key? If so, stop the loop.

107 if event.key == K\_ESCAPE:

108 running = False

109

110 # Did the user click the window close button? If so, stop the loop.

111 elif event.type == QUIT:

112 running = False

113

114 # Add a new enemy?

115 elif event.type == ADDENEMY:

116 # Create the new enemy and add it to sprite groups

117 new\_enemy = Enemy()

118 enemies.add(new\_enemy)

119 all\_sprites.add(new\_enemy)

120

121 # Get the set of keys pressed and check for user input

122 pressed\_keys = pygame. key.get\_pressed()

123 player.update(pressed\_keys)

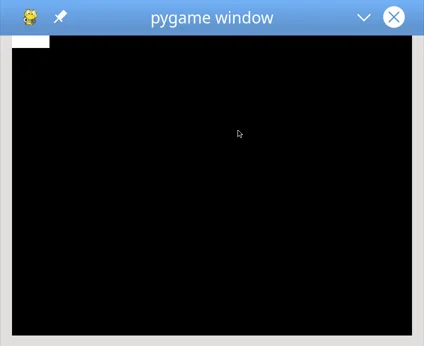
124

125 # Update enemy position

126 enemies.update()

Whenever the event handler sees the new ADDENEMY event on line 115, it creates an Enemy and adds it to enemies and all\_sprites. Since Enemy is in all\_sprites, it will get drawn in every frame. You also need to call enemies.update() on line 126, which updates everything in enemies, to ensure they move properly:

However, that’s not the only reason there’s a group of just enemies.



## **Collision Detection**

Your game design calls for the game to end whenever an enemy collides with the player. Checking for collisions is a basic technique of game programming, and usually requires some non-trivial math to determine whether two sprites will overlap each other.

This is where a framework like pygame comes in handy! Writing collision detection code is tedious, but pygame has a LOT of [collision detection methods](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_rect) available for you to use.

For this tutorial, you’ll use a method called [.spritecollideany()](https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollideany), which is read as “sprite collide any.” This method accepts a Sprite and a Group as parameters. It looks at every object in the Group and checks if its .rect intersects with the .rect of the Sprite. If so, then it returns True. Otherwise, it returns False. This is perfect for this game since you need to check if the single player collides with one of a Group of enemies.

Here’s what that looks like in the code:

130# Draw all sprites

131for entity in all\_sprites:

132 screen.blit(entity.surf, entity.rect)

133

134# Check if any enemies have collided with the player

135if pygame. sprite.spritecollideany(player, enemies):

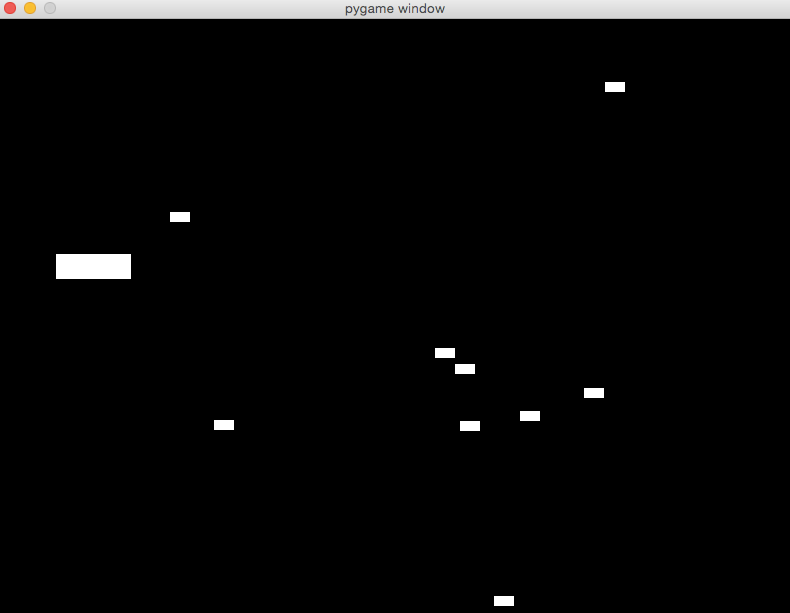
136 # If so, then remove the player and stop the loop

137 player.kill()

138 running = False

Line 135 tests whether a player has collided with any of the objects in enemies. If so, then player. kill() is called to remove it from every group to which it belongs. Since the only objects being rendered are in all\_sprites, the player will no longer be rendered. Once the player has been killed, you need to exit the game as well, so you set running = False to break out of the game loop on line 138.

At this point, you’ve got the basic elements of a game in place:



Now, let’s dress it up a bit, make it more playable, and add some advanced capabilities to help it stand out.

## **Sprite Images**

Alright, you have a game, but let’s be honest… It’s kind of ugly. The player and enemies are just white blocks on a black background. That was state-of-the-art when [Pong](https://en.wikipedia.org/wiki/Pong) was new, but it just doesn’t cut it anymore. Let’s replace all those boring white rectangles with some cooler images that will make the game feel like an actual game.

Earlier, you learned that images on disk can be loaded into a Surface with some help from the image module. For this tutorial, we made a little jet for the player and some missiles for the enemies. You’re welcome to use this art, draw your own, or download some [free game art assets](https://realpython.com/pygame-a-primer/#note-on-sources) to use. You can click the link below to download the art used in this tutorial:

### Altering the Object Constructors

Before you use images to represent the player and enemy sprites, you need to make some changes to their constructors. The code below replaces the code used previously:

7# Import pygame. locals for easier access to key coordinates

8# Updated to conform to flake8 and black standards

9# from pygame.locals import \*

10from pygame.locals import (

11 RLEACCEL,

12 K\_UP,

13 K\_DOWN,

14 K\_LEFT,

15 K\_RIGHT,

16 K\_ESCAPE,

17 KEYDOWN,

18 QUIT,

19)

20

21# Define constants for the screen width and height

22SCREEN\_WIDTH = 800

23SCREEN\_HEIGHT = 600

24

25

26# Define the Player object by extending pygame. sprite.Sprite

27# Instead of a surface, use an image for a better-looking sprite

28class Player(pygame.sprite.Sprite):

29 def \_\_init\_\_(self):

30 super(Player, self).\_\_init\_\_()

31 self.surf = pygame. image.load("jet.png").convert()

32 self. surf.set\_colorkey((255, 255, 255), RLEACCEL)

33 self.rect = self. surf.get\_rect()

Let’s unpack line 31 a bit. pygame. image.load() loads an image from the disk. You pass it a path to the file. It returns a Surface, and the .convert() call optimizes the Surface, making future .blit() calls faster.

Line 32 uses .set\_colorkey() to indicate the color pygame will render as transparent. In this case, you choose white, because that’s the background color of the jet image. The [RLEACCEL](https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set_colorkey) constant is an optional parameter that helps pygame render more quickly on non-accelerated displays. This is added to the pygame. locals import statement on line 11.

Nothing else needs to change. The image is still a Surface, except now it has a picture painted on it. You still use it in the same way.

Here’s what similar changes to the Enemy look like:

59# Define the enemy object by extending pygame. sprite.Sprite

60# Instead of a surface, use an image for a better-looking sprite

61class Enemy(pygame.sprite.Sprite):

62 def \_\_init\_\_(self):

63 super(Enemy, self).\_\_init\_\_()

64 self.surf = pygame. image.load("missile.png").convert()

65 self. surf.set\_colorkey((255, 255, 255), RLEACCEL)

66 # The starting position is randomly generated, as is the speed

67 self.rect = self. surf.get\_rect(

68 center=(

69 random.randint(SCREEN\_WIDTH + 20, SCREEN\_WIDTH + 100),

70 random.randint(0, SCREEN\_HEIGHT),

71 )

72 )

73 self.speed = random.randint(5, 20)

Running the program now should show that this is the same game you had before, except now you’ve added some nice graphics **skins** with images. But why stop at just making the player and enemy sprites look nice? Let’s add a few clouds going past to give the impression of a jet flying through the sky.

### 

### Adding Background Images

For background clouds, you use the same principles as you did for Player and Enemy:

1. Create the Cloud class.
2. Add an image of a cloud to it.
3. Create a method .update() that moves the cloud toward the left side of the screen.
4. Create a custom event and handler to create new cloud objects at a set time interval.
5. Add the newly created cloud objects to a new Group called clouds.
6. Update and draw the clouds in your game loop.

Here’s what Cloud looks like:

83# Define the cloud object by extending pygame. sprite.Sprite

84# Use an image for a better-looking sprite

85class Cloud(pygame.sprite.Sprite):

86 def \_\_init\_\_(self):

87 super(Cloud, self).\_\_init\_\_()

88 self.surf = pygame. image.load("cloud.png").convert()

89 self. surf.set\_colorkey((0, 0, 0), RLEACCEL)

90 # The starting position is randomly generated

91 self.rect = self. surf.get\_rect(

92 center=(

93 random.randint(SCREEN\_WIDTH + 20, SCREEN\_WIDTH + 100),

94 random.randint(0, SCREEN\_HEIGHT),

95 )

96 )

97

98 # Move the cloud based on a constant speed

99 # Remove the cloud when it passes the left edge of the screen

100 def update(self):

101 self.rect.move\_ip(-5, 0)

102 if self. rect.right < 0:

103 self.kill()

That should all look very familiar. It’s pretty much the same as Enemy.

To have clouds appear at certain intervals, you’ll use an event creation code similar to what you used to create new enemies. Put it right below the enemy creation event:

116# Create custom events for adding a new enemy and a cloud

117ADDENEMY = pygame.USEREVENT + 1

118pygame.time.set\_timer(ADDENEMY, 250)

119ADDCLOUD = pygame.USEREVENT + 2

120pygame.time.set\_timer(ADDCLOUD, 1000)

This says to wait 1000 milliseconds, or one second, before creating the next cloud.

Next, create a new Group to hold each newly created cloud:

125# Create groups to hold enemy sprites, cloud sprites, and all sprites

126# - enemies are used for collision detection and position updates

127# - clouds are used for position updates

128# - all\_sprites is used for rendering

129enemies = pygame. sprite.Group()

130clouds = pygame. sprite.Group()

131all\_sprites = pygame. sprite.Group()

132all\_sprites.add(player)

Next, add a handler for the new ADDCLOUD event in the event handler:

137# Main loop

138while running:

139 # Look at every event in the queue

140 for event in pygame. event.get():

141 # Did the user hit a key?

142 if event.type == KEYDOWN:

143 # Was it the Escape key? If so, then stop the loop.

144 if event.key == K\_ESCAPE:

145 running = False

146

147 # Did the user click the window close button? If so, stop the loop.

148 elif event.type == QUIT:

149 running = False

150

151 # Add a new enemy?

152 elif event.type == ADDENEMY:

153 # Create the new enemy and add it to sprite groups

154 new\_enemy = Enemy()

155 enemies.add(new\_enemy)

156 all\_sprites.add(new\_enemy)

157

158 # Add a new cloud?

159 elif event.type == ADDCLOUD:

160 # Create the new cloud and add it to sprite groups

161 new\_cloud = Cloud()

162 clouds.add(new\_cloud)

163 all\_sprites.add(new\_cloud)

Finally, make sure the clouds are updated every frame:

167# Update the position of enemies and clouds

168enemies.update()

169clouds.update()

170

171# Fill the screen with sky blue

172screen.fill((135, 206, 250))

Line 172 updates the original screen. fill() to fill the screen with a pleasant sky blue color. You can change this color to something else. Maybe you want an alien world with a purple sky, a toxic wasteland in neon green, or the surface of Mars in red!

Note that each new Cloud and Enemy are added to all\_sprites as well as clouds and enemies. This is done because each group is used for a separate purpose:

* **Rendering** is done using all\_sprites.
* **Position updates** are done using clouds and enemies.
* **Collision detection** is done using enemies.

You create multiple groups so that you can change the way sprites move or behave without impacting the movement or behavior of other sprites.

## **Game Speed**

While testing the game you may have noticed that the enemies move a little fast. If not, then that’s okay, as different machines will see different results at this point.

The reason for this is that the game loop processes frames as fast as the processor and environment will allow. Since all the sprites move once per frame, they can move hundreds of times each second. The number of frames handled each second is called the **frame rate**, and getting this right is the difference between a playable game and a forgettable one.

Normally, you want as high a frame rate as possible, but for this game, you need to slow it down a bit for the game to be playable. Fortunately, the module time contains a [Clock](https://www.pygame.org/docs/ref/time.html#pygame.time.Clock) which is designed exactly for this purpose.

Using Clock to establish a playable frame rate requires just two lines of code. The first creates a new Clock before the game loop begins:

106# Setup the clock for a decent framerate

107clock = pygame. time.Clock()

The second calls .tick() to inform pygame that the program has reached the end of the frame:

188# Flip everything to the display

189pygame.display.flip()

190

191# Ensure the program maintains a rate of 30 frames per second

192clock.tick(30)

The argument passed to .tick() establishes the desired frame rate. To do this, .tick() calculates the number of milliseconds each frame should take, based on the desired frame rate. Then, it compares that number to the number of milliseconds that have passed since the last time .tick() was called. If not enough time has passed, then .tick() delays processing to ensure that it never exceeds the specified frame rate.

Passing in a smaller frame rate will result in more time in each frame for calculations, while a larger frame rate provides smoother (and possibly faster) gameplay:



## **Sound Effects**

So far, you’ve focused on gameplay and the visual aspects of your game. Now let’s explore giving your game some auditory flavor as well. pygame provides a [mixer](https://www.pygame.org/docs/ref/mixer.html) to handle all sound-related activities. You’ll use this module’s classes and methods to provide background music and sound effects for various actions.

The name mixer refers to the fact that the module mixes various sounds into a cohesive whole. Using the [music](https://www.pygame.org/docs/ref/music.html) sub-module, you can stream individual sound files in a variety of formats, such as [MP3](https://en.wikipedia.org/wiki/MP3), [Ogg](https://en.wikipedia.org/wiki/Ogg), and [Mod](https://en.wikipedia.org/wiki/Module_file). You can also use Sound to hold a single sound effect to be played, in either Ogg or [uncompressed WAV](https://en.wikipedia.org/wiki/WAV) formats. All playback happens in the background, so when you play a Sound, the method returns immediately as the sound plays.

**Note:** The [pygame documentation](https://www.pygame.org/docs/ref/music.html) states that MP3 support is limited, and unsupported formats can cause system crashes. The sounds referenced in this article have been tested, and we recommend testing any sounds thoroughly before releasing your game.

As with most things pygame, using a mixer starts with an initialization step. Luckily, this is already handled by pygame. init(). You only need to call pygame. mixer.init() if you want to change the defaults:

106# Setup for sounds. Defaults are good.

107pygame.mixer.init()

108

109# Initialize pygame

110pygame.init()

111

112# Set up the clock for a decent framerate

113clock = pygame. time.Clock()

pygame. mixer.init() accepts several [arguments](https://www.pygame.org/docs/ref/mixer.html#pygame.mixer.init), but the defaults work fine in most cases. Note that if you want to change the defaults, you need to call pygame. mixer.init() before calling pygame.init(). Otherwise, the defaults will be in effect regardless of your changes.

After the system is initialized, you can get your sounds and background music setup:

135# Load and play background music

136# Sound source: http://ccmixter.org/files/Apoxode/59262

137# License: https://creativecommons.org/licenses/by/3.0/

138pygame.mixer.music.load("Apoxode\_-\_Electric\_1.mp3")

139pygame.mixer.music.play(loops=-1)

140

141# Load all sound files

142# Sound sources: Jon Fincher

143move\_up\_sound = pygame. mixer.Sound("Rising\_putter.ogg")

144move\_down\_sound = pygame. mixer.Sound("Falling\_putter.ogg")

145collision\_sound = pygame. mixer.Sound("Collision.ogg")

**Lines 138 and 139** load a background sound clip and begin playing it. You can tell the sound clip to loop and never end by setting the named parameter loops=-1.

**Lines 143 to 145** load three sounds you’ll use for various sound effects. The first two are rising and falling sounds, which are played when the player moves up or down. The last is the sound used whenever there is a collision. You can add other sounds as well, such as a sound for whenever an Enemy is created, or a final sound for when the game ends.

So, how do you use the sound effects? You want to play each sound when a certain event occurs. For example, when the ship moves up, you want to play move\_up\_sound. Therefore, you add a call to .play() whenever you handle that event. In the design, that means adding the following calls to .update() for Player:

26# Define the Player object by extending pygame. sprite.Sprite

27# Instead of a surface, use an image for a better-looking sprite

28class Player(pygame.sprite.Sprite):

29 def \_\_init\_\_(self):

30 super(Player, self).\_\_init\_\_()

31 self.surf = pygame. image.load("jet.png").convert()

32 self. surf.set\_colorkey((255, 255, 255), RLEACCEL)

33 self.rect = self. surf.get\_rect()

34

35 # Move the sprite based on keypresses

36 def update(self, pressed\_keys):

37 if pressed\_keys[K\_UP]:

38 self.rect.move\_ip(0, -5)

39 move\_up\_sound.play()

40 if pressed\_keys[K\_DOWN]:

41 self. rect.move\_ip(0, 5)

42 move\_down\_sound.play()

For a collision between the player and an enemy, you play the sound for when collisions are detected:

201# Check if any enemies have collided with the player

202if pygame. sprite.spritecollideany(player, enemies):

203 # If so, then remove the player

204 player.kill()

205

206 # Stop any moving sounds and play the collision sound

207 move\_up\_sound.stop()

208 move\_down\_sound.stop()

209 collision\_sound.play()

210

211 # Stop the loop

212 running = False

Here, you stop any other sound effects first, because in a collision the player is no longer moving. Then you play the collision sound and continue execution from there.

Finally, when the game is over, all sounds should stop. This is true whether the game ends due to a collision or the user exits manually. To do this, add the following lines at the end of the program after the loop:

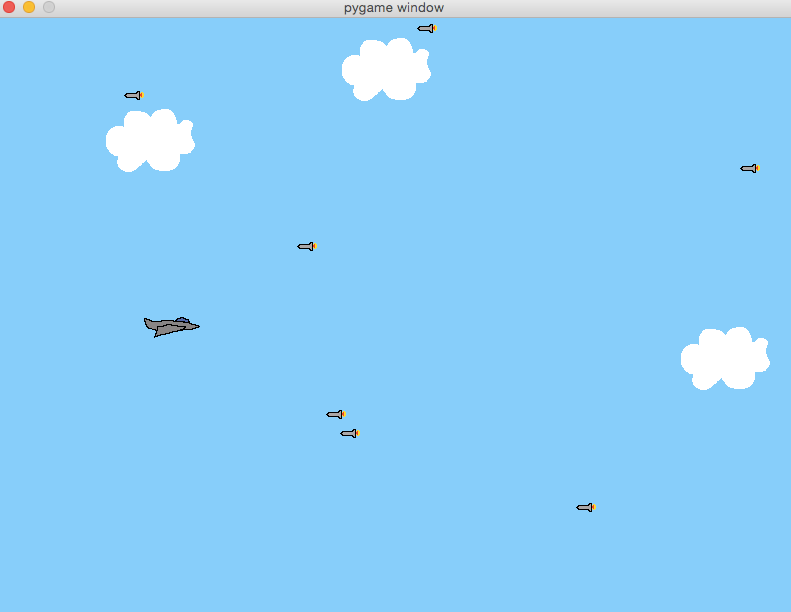
220# All done! Stop and quit the mixer.

221pygame.mixer.music.stop()

222pygame.mixer.quit()

Technically, these last few lines are not required, as the program ends right after this. However, if you decide later on to add an intro screen or an exit screen to your game, then there may be more code running after the game ends.

That’s it! Test it again, and you should see something like this:



## **A Note on Sources**

You may have noticed the comment on lines 136-137 when the background music was loaded, listing the source of the music and a link to the Creative Commons license. This was done because the creator of that sound required it. The license requirements stated that to use the sound, both proper attribution and a link to the license must be provided.

Here are some sources for music, sound, and art that you can search for useful content:

* [**OpenGameArt.org:**](https://opengameart.org/) sounds, sound effects, sprites, and other artwork
* [**Kenney.nl:**](https://kenney.nl/) sounds, sound effects, sprites, and other artwork
* [**Gamer Art 2D:**](https://www.gameart2d.com/) sprites and other artwork
* [**CC Mixter:**](http://ccmixter.org/) sounds and sound effects
* [**Freesound:**](https://freesound.org/) sounds and sound effects

As you make your games and use downloaded content such as art, music, or code from other sources, please be sure that you are complying with the licensing terms of those sources.

## 

## **Conclusion**

Throughout this tutorial, you’ve learned how game programming with pygame differs from standard procedural programming. You’ve also learned how to:

* Implement event loops
* Draw items on the screen
* Play sound effects and music
* Handle user input

To do this, you used a subset of the pygame modules, including the display, mixer and music, time, image, event, and key modules. You also used several pygame classes, including Rect, Surface, Sound, and Sprite. But these only scratch the surface of what pygame can do! Check out the [official pygame documentation](https://www.pygame.org/docs/) for a full list of available modules and classes.

You can find all of the code, graphics, and sound files for this article by clicking the link: [**Sample Code link**](https://realpython.com/bonus/pygame-sample-code/) **:** [**Click here to download the source code for the PyGame sample project**](https://realpython.com/bonus/pygame-sample-code/)