

In-depth game example

This tutorial will walk you through the steps of writing a simple Asteroids clone. It is assumed that the reader is familiar with writing and running Python programs. This is not a programming tutorial, but it should hopefully be clear enough to follow even if you're a beginner. If you get stuck, first have a look at the relevant sections of the programming guide. The full source code can also be found in the *examples/game/* folder of the *pyglet* source directory, which you can follow along with. If anything is still not clear, let us know!

Basic graphics




Lets begin! The first version of our game will simply show a score of zero, a label showing the name of the program, three randomly placed asteroids, and the player's ship. Nothing will move.

Setting up

First things first, make sure you have *pyglet* installed. Then, we will set up the folder structure for our project. Since this example game is written in stages, we will have several *version* folders at various stages of development. We will also have a shared resource folder with the images, called 'resources,' outside of the example folders. Each *version* folder contains a Python file called *asteroid.py* which runs the game, as well as a sub-folder named *game* where we will place additional modules; this is where most of the logic will be. Your folder structure should look like this:

```
game/
  resources/
    (images go here)
  version1/
    asteroid.py
    game/
      __init__.py
```

Getting a window

To set up a window, simply *import pyglet*, create a new instance of `pyglet.window`   [latest](#) 
call *pyglet.app.run()*:

```
import pygame
game_window = pygame.window.Window(800, 600)

if __name__ == '__main__':
    pygame.app.run()
```

If you run the code above, you should see a window full of junk that goes away when you press Esc. (What you are seeing is raw uninitialized graphics memory).

Loading and displaying an image

Since our images will reside in a directory other than the example's root directory, we need to tell pygame where to find them:

```
import pygame
pygame.resource.path = ['../resources']
pygame.resource.reindex()
```

pygame's `pygame.resource` module takes all of the hard work out of finding and loading game resources such as images, sounds, etc.. All that you need to do is tell it where to look, and reindex it. In this example game, the resource path starts with `../` because the resources folder is on the same level as the `version1` folder. If we left it off, pygame would look inside `version1/` for the `resources/` folder.

Now that pygame's resource module is initialized, we can easily load the images with the `image()` function of the resource module:

```
player_image = pygame.resource.image("player.png")
bullet_image = pygame.resource.image("bullet.png")
asteroid_image = pygame.resource.image("asteroid.png")
```

Centering the images

Pygame will draw and position all images from their lower left corner by default. We don't want this behavior for our images, which need to rotate around their centers. All we have to do to achieve this is to set their anchor points. Lets create a function to simplify this:

```
def center_image(image):  
    """Sets an image's anchor point to its center"""  
    image.anchor_x = image.width // 2  
    image.anchor_y = image.height // 2
```

Now we can just call `center_image()` on all of our loaded images:

```
center_image(player_image)  
center_image(bullet_image)  
center_image(asteroid_image)
```

Remember that the `center_image()` function must be defined before it can be called at the module level. Also, note that zero degrees points directly to the right in `pyglet`, so the images are all drawn with their front pointing to the right.

To access the images from `asteroid.py`, we need to use something like *from game import resources*, which we'll get into in the next section.

Initializing objects

We want to put some labels at the top of the window to give the player some information about the score and the current difficulty level. Eventually, we will have a score display, the name of the level, and a row of icons representing the number of remaining lives.


Making the labels

To make a text label in `pyglet`, just initialize a `pyglet.text.Label` object:

```
score_label = pyglet.text.Label(text="Score: 0", x=10, y=575)  
level_label = pyglet.text.Label(text="My Amazing Game",  
                                x=game_window.width//2, y=575, anchor_x='center')
```

Notice that the second label is centered using the `anchor_x` attribute.

Drawing the labels

We want `pyglet` to run some specific code whenever the window is drawn. An `on_draw()` event is dispatched to the window to give it a chance to redraw its contents. `pyglet` provides several ways to attach event handlers to objects; a simple way is to use a decorator:  [latest](#) ▼

```
@game_window.event
def on_draw():
    # draw things here
```

The `@game_window.event` decorator lets the `Window` instance know that our `on_draw()` function is an event handler. The `on_draw()` event is fired whenever - you guessed it - the window needs to be redrawn. Other events include `on_mouse_press()` and `on_key_press()`.

Now we can fill the method with the functions necessary to draw our labels. Before we draw anything, we should clear the screen. After that, we can simply call each object's `draw()` function:

```
@game_window.event
def on_draw():
    game_window.clear()

    level_label.draw()
    score_label.draw()
```

Now when you run `asteroid.py`, you should get a window with a score of zero in the upper left corner and a centered label reading "My Amazing Game" at the top of the screen.


Making the player and asteroid sprites

The player should be an instance or subclass of `pygame.sprite.Sprite`, like so:

```
from game import resources
...
player_ship = pygame.sprite.Sprite(img=resources.player_image, x=400, y=300)
```

To get the player to draw on the screen, add a line to `on_draw()`:

```
@game_window.event
def on_draw():
    ...
    player_ship.draw()
```

Loading the asteroids is a little more complicated, since we'll need to place more than one at random locations that don't immediately collide with the player. Let's put the load new game submodule called `load.py`:  [latest](#) ▼

```

import pygame
import random
from . import resources

def asteroids(num_asteroids):
    asteroids = []
    for i in range(num_asteroids):
        asteroid_x = random.randint(0, 800)
        asteroid_y = random.randint(0, 600)
        new_asteroid = pygame.sprite.Sprite(img=resources.asteroid_image,
                                             x=asteroid_x, y=asteroid_y)
        new_asteroid.rotation = random.randint(0, 360)
        asteroids.append(new_asteroid)
    return asteroids

```

All we are doing here is making a few new sprites with random positions. There's still a problem, though - an asteroid might randomly be placed exactly where the player is, causing immediate death. To fix this issue, we'll need to be able to tell how far away new asteroids are from the player. Here is a simple function to calculate that distance:

```

import math
...
def distance(point_1=(0, 0), point_2=(0, 0)):
    """Returns the distance between two points"""
    return math.sqrt((point_1[0] - point_2[0]) ** 2 + (point_1[1] - point_2[1]) ** 2)

```

To check new asteroids against the player's position, we need to pass the player's position into the `asteroids()` function and keep regenerating new coordinates until the asteroid is far enough away. pygame sprites keep track of their position both as a tuple (`Sprite.position`) and as x, y, and z attributes (`Sprite.x`, `Sprite.y`, `Sprite.z`). To keep our code short, we'll just pass the position tuple into the function. We're not using the z value, so we just use a throwaway variable for that:

```

def asteroids(num_asteroids, player_position):
    asteroids = []
    for i in range(num_asteroids):
        asteroid_x, asteroid_y, _ = player_position
        while distance((asteroid_x, asteroid_y), player_position) < 100:
            asteroid_x = random.randint(0, 800)
            asteroid_y = random.randint(0, 600)
        new_asteroid = pygame.sprite.Sprite(
            img=resources.asteroid_image, x=asteroid_x, y=asteroid_y)
        new_asteroid.rotation = random.randint(0, 360)
        asteroids.append(new_asteroid)
    return asteroids

```

For each asteroid, it chooses random positions until it finds one away from the player, creates the sprite, and gives it a random rotation. Each asteroid is appended to a list, which is returned.

Now you can load three asteroids like this:

```
from game import resources, load
...
asteroids = load.asteroids(3, player_ship.position)
```

The asteroids variable now contains a list of sprites. Drawing them on the screen is as simple as it was for the player's ship - just call their `draw()` methods:

```
@game_window.event
def on_draw():
    ...
    for asteroid in asteroids:
        asteroid.draw()
```

This wraps up the first section. Your “game” doesn’t do much of anything yet, but we’ll get to that in the following sections. You may want to look over the *examples/game/version1* folder in the pygame source to review what we’ve done, and to find a functional copy.

Basic motion

In the second version of the example, we’ll introduce a simpler, faster way to draw all of the game objects, as well as add row of icons indicating the number of lives left. We’ll also write some code to make the player and the asteroids obey the laws of physics.

Drawing with batches

Calling each object’s `draw()` method manually can become cumbersome and tedious if there are many different kinds of objects. It’s also very inefficient if you need to draw a large number of objects. The pygame `pygame.graphics.Batch` class simplifies drawing by letting you draw all your objects with a single function call. All you need to do is create a batch, pass it into each object you want to draw, and call the batch’s `draw()` method.

To create a new batch, simply create an instance of `pygame.graphics.Batch`:

```
main_batch = pygame.graphics.Batch()
```

To make an object a member of a batch, just pass the batch into its constructor as the batch keyword argument:

```
score_label = pygame.text.Label(text="Score: 0", x=10, y=575, batch=main_batch)
```

Add the batch keyword argument to each graphical object created in `asteroid.py`.

To use the batch with the asteroid sprites, we'll need to pass the batch into the `game.load.asteroid()` function, then just add it as a keyword argument to each new sprite. Update the function:

```
def asteroids(num_asteroids, player_position, batch=None):
    ...
    new_asteroid = pygame.sprite.Sprite(img=resources.asteroid_image,
                                         x=asteroid_x, y=asteroid_y,
                                         batch=batch)
```

And update the place where it's called:

```
asteroids = load.asteroids(3, player_ship.position, main_batch)
```

Now you can replace those five lines of `draw()` calls with just one:

```
main_batch.draw()
```

Now when you run `asteroid.py`, it should look exactly the same.

Displaying little ship icons

To show how many lives the player has left, we'll need to draw a little row of icons in the upper right corner of the screen. Since we'll be making more than one using the same template, let's create a function called `player_lives()` in the `load` module to generate them. The icons should look the same as the player's ship. We could create a scaled version using an image editor, or we could just let pygame do the scaling. I don't know about you, but I prefer the option that requires less work.

 [latest](#) ▼

The function for creating the icons is almost exactly the same as the one for creating asteroids. For each icon we just create a sprite, give it a position and scale, and append it to the return list:

```
def player_lives(num_icons, batch=None):
    player_lives = []
    for i in range(num_icons):
        new_sprite = pygame.sprite.Sprite(img=resources.player_image,
                                           x=785-i*30, y=585, batch=batch)
        new_sprite.scale = 0.5
        player_lives.append(new_sprite)
    return player_lives
```

The player icon is 50x50 pixels, so half that size will be 25x25. We want to put a little bit of space between each icon, so we create them at 30-pixel intervals starting from the right side of the screen and moving to the left. Note that like the *asteroids()* function, *player_lives()* takes a *batch* argument.

Making things move

The game would be pretty boring if nothing on the screen ever moved. To achieve motion, we'll need to write our own set of classes to handle frame-by-frame movement calculations. We'll also need to write a Player class to respond to keyboard input.

Creating the basic motion class

Since every visible object is represented by at least one Sprite, we may as well make our basic motion class a subclass of `pygame.sprite.Sprite`. Another approach would be to have our class have a `sprite` attribute.

Create a new game submodule called `physicalobject.py` and declare a `PhysicalObject` class. The only new attributes we'll be adding will store the object's velocity, so the constructor will be simple:

```
class PhysicalObject(pygame.sprite.Sprite):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.velocity_x, self.velocity_y = 0.0, 0.0
```

Each object will need to be updated every frame, so let's write an *update()* method:

```
def update(self, dt):
    self.x += self.velocity_x * dt
    self.y += self.velocity_y * dt
```


What's `dt`? It's the "delta time", or "time step". Game frames are not instantaneous, and they don't always take equal amounts of time to draw. If you've ever tried to play a modern game on an old machine, you know that frame rates can jump all over the place. There are a number of ways to deal with this problem, the simplest one being to just multiply all time-sensitive operations by `dt`. I'll show you how this value is calculated later.

If we give objects a velocity and just let them go, they will fly off the screen before long. Since we're making an Asteroids clone, we would rather they just wrapped around the screen. Here is a simple function that accomplishes the goal:

```
def check_bounds(self):
    min_x = -self.image.width / 2
    min_y = -self.image.height / 2
    max_x = 800 + self.image.width / 2
    max_y = 600 + self.image.height / 2
    if self.x < min_x:
        self.x = max_x
    elif self.x > max_x:
        self.x = min_x
    if self.y < min_y:
        self.y = max_y
    elif self.y > max_y:
        self.y = min_y
```

As you can see, it simply checks to see if objects are no longer visible on the screen, and if so, it moves them to the other side of the screen. To make every `PhysicalObject` use this behavior, add a call to `self.check_bounds()` at the end of `update()`.

To make the asteroids use our new motion code, just import the `physicalobject` module and change the `new_asteroid = ...` line to create a new `PhysicalObject` instead of a `Sprite`. You'll also want to give them a random initial velocity. Here is the new, improved `load.asteroids()` function:

```
def asteroids(num_asteroids, player_position, batch=None):
    ...
    new_asteroid = physicalobject.PhysicalObject(...)
    new_asteroid.rotation = random.randint(0, 360)
    new_asteroid.velocity_x = random.random()*40
    new_asteroid.velocity_y = random.random()*40
    ...
```

Writing the game update function

To call each object's `update()` method every frame, we first need to have a list of `PhysicalObject`s. For now, we can just declare it after setting up all the other objects:

```
game_objects = [player_ship] + asteroids
```

Now we can write a simple function to iterate over the list:

```
def update(dt):  
    for obj in game_objects:  
        obj.update(dt)
```

The *update()* function takes a *dt* parameter because it is still not the source of the actual time step.

Calling the update() function

We need to update the objects at least once per frame. What's a frame? Well, most screens have a maximum refresh rate of 60 hertz. If we set our loop to run at exactly 60 hertz, though, the motion will look a little jerky because it won't match the screen exactly. Instead, we can have it update twice as fast, 120 times per second, to get smooth animation.

The best way to call a function 120 times per second is to ask pygame to do it. The `pygame.clock` module contains a number of ways to call functions periodically or at some specified time in the future. The one we want is `pygame.clock.schedule_interval()`:

```
pygame.clock.schedule_interval(update, 1/120.0)
```

Putting this line above *pygame.app.run()* in the `if __name__ == '__main__':` block tells pygame to call *update()* 120 times per second. Pygame will pass in the elapsed time, i.e. *dt*, as the only parameter.

Now when you run *asteroid.py*, you should see your formerly static asteroids drifting serenely across the screen, reappearing on the other side when they slide off the edge.

Writing the Player class

In addition to obeying the basic laws of physics, the player object needs to respond to keyboard input. Start by creating a *game.player* module, importing the appropriate modules, and subclassing *PhysicalObject*:

```
from . import physicalobject, resources

class Player(physicalobject.PhysicalObject):

    def __init__(self, *args, **kwargs):
        super().__init__(img=resources.player_image, *args, **kwargs)
```

So far, the only difference between a Player and a PhysicalObject is that a Player will always have the same image. But Player objects need a couple more attributes. Since the ship will always thrust with the same force in whatever direction it points, we'll need to define a constant for the magnitude of that force. We should also define a constant for the ship's rotation speed:

```
self.thrust = 300.0
self.rotate_speed = 200.0
```

Now we need to get the class to respond to user input. Pyglet uses an event-based approach to input, sending key press and key release events to registered event handlers. But we want to use a polling approach in this example, checking periodically if a key is down. One way to accomplish that is to maintain a dictionary of keys. First, we need to initialize the dictionary in the constructor:

```
self.keys = dict(left=False, right=False, up=False)
```

Then we need to write two methods, *on_key_press()* and *on_key_release()*. When pyglet checks a new event handler, it looks for these two methods, among others:

```

import math
from pygamelet.window import key
from . import physicalobject, resources

class Player(physicalobject.PhysicalObject):

    def on_key_press(self, symbol, modifiers):
        if symbol == key.UP:
            self.keys['up'] = True
        elif symbol == key.LEFT:
            self.keys['left'] = True
        elif symbol == key.RIGHT:
            self.keys['right'] = True

    def on_key_release(self, symbol, modifiers):
        if symbol == key.UP:
            self.keys['up'] = False
        elif symbol == key.LEFT:
            self.keys['left'] = False
        elif symbol == key.RIGHT:
            self.keys['right'] = False

```

That looks pretty cumbersome. There's a better way to do it which we'll see later, but for now, this version serves as a good demonstration of pygamelet's event system.

The last thing we need to do is write the *update()* method. It follows the same behavior as a *PhysicalObject* plus a little extra, so we'll need to call *PhysicalObject*'s *update()* method and then respond to input:

```

def update(self, dt):
    super().update(dt)

    if self.keys['left']:
        self.rotation -= self.rotate_speed * dt
    if self.keys['right']:
        self.rotation += self.rotate_speed * dt

```

Pretty simple so far. To rotate the player, we just add the rotation speed to the angle, multiplied by *dt* to account for time. Note that *Sprite* objects' rotation attributes are in degrees, with clockwise as the positive direction. This means that you need to call *math.degrees()* or *math.radians()* and make the result negative whenever you use Python's built-in math functions with the *Sprite* class, since those functions use radians instead of degrees, and their positive direction is counter-clockwise. The code to make the ship thrust forward uses an example of such a conversion:

```

if self.keys['up']:
    angle_radians = -math.radians(self.rotation)
    force_x = math.cos(angle_radians) * self.thrust * dt
    force_y = math.sin(angle_radians) * self.thrust * dt
    self.velocity_x += force_x
    self.velocity_y += force_y

```

First, we convert the angle to radians so that *math.cos()* and *math.sin()* will get the correct values. Then we apply some simple physics to modify the ship's X and Y velocity components and push the ship in the right direction.

We now have a complete Player class. If we add it to the game and tell pyglet that it's an event handler, we should be good to go.

Integrating the player class

The first thing we need to do is make *player_ship* an instance of *Player*:

```

from game import player
...
player_ship = player.Player(x=400, y=300, batch=main_batch)

```

Now we need to tell pyglet that *player_ship* is an event handler. To do that, we need to push it onto the event stack with *game_window.push_handlers()*:

```

game_window.push_handlers(player_ship)

```

That's it! Now you should be able to run the game and move the player with the arrow keys.

Giving the player something to do

In any good game, there needs to be something working against the player. In the case of *Asteroids*, it's the threat of collision with, well, an asteroid. Collision detection requires a lot of infrastructure in the code, so this section will focus on making it work. We'll also clean up the player class and show some visual feedback for thrusting.

Simplifying player input

Right now, the *Player* class handles all of its own keyboard events. It spends 13 lines of code doing nothing but setting boolean values in a dictionary. One would think that there would be a better way, and there is: `pyglet.window.key.KeyStateHandler`. This handy class automatically

does what we have been doing manually: it tracks the state of every key on the keyboard.

To start using it, we need to initialize it and push it onto the event stack instead of the `Player` class. First, let's add it to `Player`'s constructor:

```
self.key_handler = key.KeyStateHandler()
```

We also need to push the `key_handler` object onto the event stack. Keep pushing the `player_ship` object in addition to its key handler, because we'll need it to keep handling key press and release events later:

```
game_window.push_handlers(player_ship.key_handler)
```

Since `Player` now relies on `key_handler` to read the keyboard, we need to change the `update()` method to use it. The only changes are in the if conditions:

```
if self.key_handler[key.LEFT]:
    ...
if self.key_handler[key.RIGHT]:
    ...
if self.key_handler[key.UP]:
    ...
```

Now we can remove the `on_key_press()` and `on_key_release()` methods from the class. It's just that simple. If you need to see a list of key constants, you can check the API documentation under

`pygame.window.key`.

Adding an engine flame

Without visual feedback, it can be difficult to tell if the ship is actually thrusting forward or not, especially for an observer just watching someone else play the game. One way to provide visual feedback is to show an engine flame behind the player while the player is thrusting.

Loading the flame image

The player will now be made of two sprites. There's nothing preventing us from letting a `Sprite` own another `Sprite`, so we'll just give `Player` an `engine_sprite` attribute and update frame. For our purposes, this approach will be the simplest and most scalable.

To make the flame draw in the correct position, we could either do some complicated math every frame, or we could just move the image's anchor point. First, load the image in `resources.py`:

```
engine_image = pygame.image.load("engine_flame.png")
```

To get the flame to draw behind the player, we need to move the flame image's center of rotation to the right, past the end of the image. To do that, we just set its `anchor_x` and `anchor_y` attributes:

```
engine_image.anchor_x = engine_image.width * 1.5  
engine_image.anchor_y = engine_image.height / 2
```

Now the image is ready to be used by the player class. If you're still confused about anchor points, experiment with the values for `engine_image`'s anchor point when you finish this section.

Creating and drawing the flame

The engine sprite needs to be initialized with all the same arguments as `Player`, except that it needs a different image and must be initially invisible. The code for creating it belongs in `Player.__init__()` and is very straightforward:

```
self.engine_sprite = pygame.sprite.Sprite(img=resources.engine_image, *args, **kwargs)  
self.engine_sprite.visible = False
```

To make the engine sprite appear only while the player is thrusting, we need to add some logic to the `if self.key_handler[key.UP]` block in the `update()` method:

```
if self.key_handler[key.UP]:  
    ...  
    self.engine_sprite.visible = True  
else:  
    self.engine_sprite.visible = False
```

To make the sprite appear at the player's position, we also need to update its position and rotation attributes:

```
if self.key_handler[key.UP]:
    ...
    self.engine_sprite.rotation = self.rotation
    self.engine_sprite.x = self.x
    self.engine_sprite.y = self.y
    self.engine_sprite.visible = True
else:
    self.engine_sprite.visible = False
```

Cleaning up after death

When the player is inevitably smashed to bits by an asteroid, he will disappear from the screen. However, simply removing the Player instance from the `game_objects` list is not enough for it to be removed from the graphics batch. To do that, we need to call its `delete()` method. Normally a Sprite's own `delete()` method will work fine without modifications, but our subclass has its own child Sprite (the engine flame) which must also be deleted when the Player instance is deleted. To get both to die gracefully, we must write a simple but slightly enhanced `delete()` method:

```
def delete(self):
    self.engine_sprite.delete()
    super().delete()
```

The Player class is now cleaned up and ready to go.

Checking For collisions

To make objects disappear from the screen, we'll need to manipulate the game objects list. Every object will need to check every other object's position against its own, and each object will have to decide whether or not it should be removed from the list. The game loop will then check for dead objects and remove them from the list.

Checking all object pairs

We need to check every object against every other object. The simplest method is to use nested loops. This method will be inefficient for a large number of objects, but it will work for our purposes. We can use one easy optimization and avoid checking the same pair of objects twice. Here's the setup for the loops, which belongs in `update()`. It simply iterates over all object pairs without doing anything:


```
for i in range(len(game_objects)):
    for j in range(i+1, len(game_objects)):
        obj_1 = game_objects[i]
        obj_2 = game_objects[j]
```

We'll need a way to check if an object has already been killed. We could go over to `PhysicalObject` right now and put it in, but let's keep working on the game loop and implement the method later. For now, we'll just assume that everything in `game_objects` has a `dead` attribute which will be `False` until the class sets it to `True`, at which point it will be ignored and eventually removed from the list.

To perform the actual check, we'll also need to call two more methods that don't exist yet. One method will determine if the two objects actually collide, and the other method will give each object an opportunity to respond to the collision. The checking code itself is easy to understand, so I won't bother you with further explanations:

```
if not obj_1.dead and not obj_2.dead:
    if obj_1.collides_with(obj_2):
        obj_1.handle_collision_with(obj_2)
        obj_2.handle_collision_with(obj_1)
```

Now all that remains is for us to go through the list and remove dead objects:

```
for to_remove in [obj for obj in game_objects if obj.dead]:
    to_remove.delete()
    game_objects.remove(to_remove)
```

As you can see, it simply calls the object's `delete()` method to remove it from any batches, then it removes it from the list. If you haven't used list comprehensions much, the above code might look like it's removing objects from the list while traversing it. Fortunately, the list comprehension is evaluated before the loop actually runs, so there should be no problems.

Implementing the collision functions

We need to add three things to the `PhysicalObject` class: the `dead` attribute, the `collides_with()` method, and the `handle_collision_with()` method. The `collides_with()` method will need to use the `distance()` function, so let's start by moving that function into its own submodule of `game`, called `util.py`:

```
import pygame, math

def distance(point_1=(0, 0), point_2=(0, 0)):
    return math.sqrt(
        (point_1[0] - point_2[0]) ** 2 +
        (point_1[1] - point_2[1]) ** 2)
```

Remember to call from util import distance in load.py. Now we can write *PhysicalObject.collides_with()* without duplicating code:

```
def collides_with(self, other_object):
    collision_distance = self.image.width/2 + other_object.image.width/2
    actual_distance = util.distance(self.position, other_object.position)

    return (actual_distance <= collision_distance)
```

The collision handler function is even simpler, since for now we just want every object to die as soon as it touches another object:

```
def handle_collision_with(self, other_object):
    self.dead = True
```

One last thing: set `self.dead = False` in `PhysicalObject.__init__()`.



And that's it! You should be able to zip around the screen, engine blazing away. If you hit something, both you and the thing you collided with should disappear from the screen. There's still no game, but we are clearly making progress.

Collision response

In this section, we'll add bullets. This new feature will require us to start adding things to the `game_objects` list during the game, as well as have objects check each others' types to make a decision about whether or not they should die.

Adding objects during play

How?

We handled object removal with a boolean flag. Adding objects will be a little bit  [latest](#)  complicated. For one thing, an object can't just say "Add me to the list!" It has to come from somewhere. For another thing, an object might want to add more than one other object at a

time.

There are a few ways to solve this problem. To avoid circular references, keep our constructors nice and short, and avoid adding extra modules, we'll have each object keep a list of new child objects to be added to `game_objects`. This approach will make it easy for any object in the game to spawn more objects.

Tweaking the game loop

The simplest way to check objects for children and add those children to the list is to add two lines of code to the `game_objects` loop. We haven't implemented the `new_objects` attribute yet, but when we do, it will be a list of objects to add:

```
for obj in game_objects:
    obj.update(dt)
    game_objects.extend(obj.new_objects)
    obj.new_objects = []
```

Unfortunately, this simple solution is problematic. It's generally a bad idea to modify a list while iterating over it. The fix is to simply add new objects to a separate list, then add the objects in the separate list to `game_objects` after we have finished iterating over it.

Declare a `to_add` list just above the loop and add new objects to it instead. At the very bottom of `update()`, after the object removal code, add the objects in `to_add` to `game_objects`:

```
...collision...

to_add = []

for obj in game_objects:
    obj.update(dt)
    to_add.extend(obj.new_objects)
    obj.new_objects = []

...removal...

game_objects.extend(to_add)
```

Putting the attribute in `PhysicalObject`

As mentioned before, all we have to do is declare a `new_objects` attribute in the `PhysicalObject` class:

```
def __init__(self, *args, **kwargs):
    ....
    self.new_objects = []
```

To add a new object, all we have to do is put something in `new_objects`, and the main loop will see it, add it to the `game_objects` list, and clear `new_objects`.

Adding bullets

Writing the bullet class

For the most part, bullets act like any other `PhysicalObject`, but they have two differences, at least in this game: they only collide with some objects, and they disappear from the screen after a couple of seconds to prevent the player from flooding the screen with bullets.

First, make a new submodule of game called `bullet.py` and start a simple subclass of `PhysicalObject`:

```
import pygame
from . import physicalobject, resources

class Bullet(physicalobject.PhysicalObject):
    """Bullets fired by the player"""

    def __init__(self, *args, **kwargs):
        super().__init__(
            resources.bullet_image, *args, **kwargs)
```

To get bullets to disappear after a time, we could keep track of our own age and lifespan attributes, or we could let `pygame` do all the work for us. I don't know about you, but I prefer the second option. First, we need to write a function to call at the end of a bullet's life:

```
def die(self, dt):
    self.dead = True
```

Now we need to tell `pygame` to call it after half a second or so. We can do this as soon as the object is initialized by adding a call to `pygame.clock.schedule_once()` to the constructor:

```
def __init__(self, *args, **kwargs):
    super().__init__(resources.bullet_image, *args, **kwargs)
    pygame.clock.schedule_once(self.die, 0.5)
```

There's still more work to be done on the Bullet class, but before we do any more work on the class itself, let's get them on the screen.

Firing bullets

The Player class will be the only class that fires bullets, so let's open it up, import the bullet module, and add a `bullet_speed` attribute to its constructor:

```
...
from . import bullet

class Player(physicalobject.PhysicalObject):
    def __init__(self, *args, **kwargs):
        super().__init__(img=resources.player_image, *args, **kwargs)
        ...
        self.bullet_speed = 700.0
```

Now we can write the code to create a new bullet and send it hurling off into space. First, we need to resurrect the `on_key_press()` event handler:

```
def on_key_press(self, symbol, modifiers):
    if symbol == key.SPACE:
        self.fire()
```

The `fire()` method itself will be a bit more complicated. Most of the calculations will be very similar to the ones for thrusting, but there will be some differences. We'll need to spawn the bullet out at the nose of the ship, not at its center. We'll also need to add the ship's existing velocity to the bullet's new velocity, or the bullets will end up going slower than the ship if the player gets going fast enough.

As usual, convert to radians and reverse the direction:

```
def fire(self):
    angle_radians = -math.radians(self.rotation)
```

Next, calculate the bullet's position and instantiate it:

```
ship_radius = self.image.width/2
bullet_x = self.x + math.cos(angle_radians) * ship_radius
bullet_y = self.y + math.sin(angle_radians) * ship_radius
new_bullet = bullet.Bullet(bullet_x, bullet_y, batch=self.batch)
```

Set its velocity using almost the same equations:

```
bullet_vx = (
    self.velocity_x +
    math.cos(angle_radians) * self.bullet_speed
)
bullet_vy = (
    self.velocity_y +
    math.sin(angle_radians) * self.bullet_speed
)
new_bullet.velocity_x = bullet_vx
new_bullet.velocity_y = bullet_vy
```

Finally, add it to the `new_objects` list so that the main loop will pick it up and add it to `game_objects`:

```
self.new_objects.append(new_bullet)
```

At this point, you should be able to fire bullets out of the front of your ship. There's just one problem: as soon as you fire, your ship disappears. You may have noticed earlier that asteroids also disappear when they touch each other. To fix this problem, we'll need to start customizing each class's `handle_collision_with()` method.

Customizing collision behavior

There are five kinds of collisions in the current version of the game: bullet-asteroid, bullet-player, asteroid-player, bullet-bullet, and asteroid-asteroid. There would be many more in a more complex game.

In general, objects of the same type should not be destroyed when they collide, so we can generalize that behavior in `PhysicalObject`. Other interactions will require a little more work.

Letting twins ignore each other

To let two asteroids or two bullets pass each other by without a word of acknowledgment (or a dramatic explosion), we just need to check if their classes are equal in the `PhysicalObject.handle_collision_with()` method:

```
def handle_collision_with(self, other_object):
    if other_object.__class__ == self.__class__:
        self.dead = False
    else:
        self.dead = True
```

There are a few other, more elegant ways to check for object equality in Python, but the above code gets the job done.

Customizing bullet collisions

Since bullet collision behavior can vary so wildly across objects, let's add a `reacts_to_bullets` attribute to `PhysicalObjects` which the `Bullet` class can check to determine if it should register a collision or not. We should also add an `is_bullet` attribute so we can check the collision properly from both objects.

(These are not “good” design decisions, but they will work.)

First, initialize the `reacts_to_bullets` attribute to `True` in the `PhysicalObject` constructor:

```
class PhysicalObject(pygame.sprite.Sprite):
    def __init__(self, *args, **kwargs):
        ...
        self.reacts_to_bullets = True
        self.is_bullet = False
        ...

class Bullet(PhysicalObject.PhysicalObject):
    def __init__(self, *args, **kwargs):
        ...
        self.is_bullet = True
```

Then, insert a bit of code in `PhysicalObject.collides_with()` to ignore bullets under the right circumstances:

```
def collides_with(self, other_object):
    if not self.reacts_to_bullets and other_object.is_bullet:
        return False
    if self.is_bullet and not other_object.reacts_to_bullets:
        return False
    ...
```

Finally, set `self.reacts_to_bullets = False` in `Player.__init__()`. The `Bullet` class is completely finished! Now let's make something happen when a bullet hits an asteroid.

Making asteroids explode

Asteroids is challenging to players because every time you shoot an asteroid, it turns into more asteroids. We need to mimic that behavior if we want our game to be any fun. We've already done most of the hard parts. All that remains is to make another subclass of `PhysicalObject` and write a custom `handle_collision_with()` method, along with a couple of maintenance tweaks.

Writing the asteroid class

Create a new submodule of game called `asteroid.py`. Write the usual constructor to pass a specific image to the superclass, passing along any other parameters:

```
import pygame
from . import resources, physicalobject

class Asteroid(physicalobject.PhysicalObject):
    def __init__(self, *args, **kwargs):
        super().__init__(resources.asteroid_image, *args, **kwargs)
```

Now we need to write a new `handle_collision_with()` method. It should create a random number of new, smaller asteroids with random velocities. However, it should only do that if it's big enough. An asteroid should divide at most twice, and if we scale it down by half each time, then an asteroid should stop dividing when it's 1/4 the size of a new asteroid.

We want to keep the old behavior of ignoring other asteroids, so start the method with a call to the superclass's method:

```
def handle_collision_with(self, other_object):
    super().handle_collision_with(other_object)
```

Now we can say that if it's supposed to die, and it's big enough, then we should create two or three new asteroids with random rotations and velocities. We should add the old asteroid's velocity to the new ones to make it look like they come from the same object:


```

import random

class Asteroid:
    def handle_collision_with(self, other_object):
        super().handle_collision_with(other_object)
        if self.dead and self.scale > 0.25:
            num_asteroids = random.randint(2, 3)
            for i in range(num_asteroids):
                new_asteroid = Asteroid(x=self.x, y=self.y, batch=self.batch)
                new_asteroid.rotation = random.randint(0, 360)
                new_asteroid.velocity_x = (random.random() * 70 + self.velocity_x)
                new_asteroid.velocity_y = (random.random() * 70 + self.velocity_y)
                new_asteroid.scale = self.scale * 0.5
                self.new_objects.append(new_asteroid)

```

While we're here, let's add a small graphical touch to the asteroids by making them rotate a little. To do that, we'll add a `rotate_speed` attribute and give it a random value. Then we'll write an `update()` method to apply that rotation every frame.

Add the attribute in the constructor:

```

def __init__(self, *args, **kwargs):
    super().__init__(resources.asteroid_image, *args, **kwargs)
    self.rotate_speed = random.random() * 100.0 - 50.0

```

Then write the `update()` method:

```

def update(self, dt):
    super().update(dt)
    self.rotation += self.rotate_speed * dt

```

The last thing we need to do is go over to `load.py` and have the `asteroid()` method create a new `Asteroid` instead of a `PhysicalObject`:

```

from . import asteroid

def asteroids(num_asteroids, player_position, batch=None):
    ...
    for i in range(num_asteroids):
        ...
        new_asteroid = asteroid.Asteroid(x=asteroid_x, y=asteroid_y, batch=batch)
        ...
    return asteroids

```

Now we're looking at something resembling a game. It's simple, but all of the basics are there.

Next steps

So instead of walking you through a standard refactoring session, I'm going to leave it as an exercise for you to do the following:

- * Make the Score counter mean something
- * Let the player restart the level if they die
- * Implement lives and a "Game Over" screen
- * Add particle effects

Good luck! With a little effort, you should be able to figure out most of these things on your own. If you have trouble, join us on the pyglet [Discord server](#).

Maintain search performance with GenAI. [Read On](#)

Ads by EthicalAds