OBJECT ORIENTED PROGRAMMING WITH PYTHON

PROJECT BASED

Dr. Ann Al-Kazzaz

2nd semester (Lect12)

MAKING SURE TO SCORE ALL HITS

- As currently written, our code could miss scoring for some viruses. For example, if two laser beams collide with viruses during the same pass through the loop or if we make an extra-wide laser_beam to hit multiple viruses, the player will only receive points for hitting one of the viruses. To fix this, let's refine the way that laser_beam and virus collisions are detected.

- In **_check_laser_beam_virus_collisions()**, any laser_beam that collides with a virus becomes a key in the collisions dictionary. The value associated with each laser_beam is a list of viruses it has collided with. We loop through the values in the collisions dictionary to make sure we award points for each virus hit:

virus_invasion.py

```
def _check_laser_beam_virus_collisions(self):
    --snip--
    if collisions:
        for viruses in collisions.values():
            self.stats.score += self.settings.virus_points * len(viruses)
        self.sb.prep_score()
    --snip--
```

- If the collisions dictionary has been defined, we loop through all values in the dictionary. Remember that each value is a list of viruses hit by a single laser_beam. We multiply the value of each virus by the number of viruses in each list and add this amount to the current score. To test this, **change the width of a laser_beam to 300 pixels and verify that you receive points for each virus you hit with your extra-wide laser beams; then return the laser_beam width to its normal value**

INCREASING POINT VALUES

Because the game gets more difficult each time a player reaches a new level, viruses in later levels should be worth more points. To implement this functionality, we'll add code to increase the point value when the game's speed increases:

setting.py

```
Class Settings:
       def __init__(self):
             --snip--
              # How quickly the game speeds up self.speedup_scale = 1.1
1             # How quickly the virus point values increase
              self.score_scale = 1.5
              self.initialize_dynamic_settings()

      def initialize_dynamic_settings(self):
            --snip—

      def increase_speed(self):
            """Increase speed settings and virus point values."""
2           self.robot_speed *= self.speedup_scale
            self.laser_beam_speed *= self.speedup_scale
            self.virus_speed *= self.speedup_scale
            self.virus_points = int(self.virus_points * self.score_scale)
```

1 We define a rate at which points increase, which we call **score_scale** . A small increase in speed (1.1) makes the game more challenging quickly. But to see a more notable difference in scoring, we need to change the virus point value by a larger amount (1.5)

2 Now when we increase the game's speed, we also increase the point value of each hit . We use the **int()** function to increase the point value by whole integers.

ROUNDING THE SCORE

Most arcade-style shooting games report scores as multiples of 10, so let's follow that lead with our scores. Also, let's format the score to include comma separators in large numbers. We'll make this change in

scoreboard.py

```
def prep_score(self):
        """Turn the score into a rendered image."""
1       rounded_score = round(self.stats.score, -1)
2       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True,
                        self.text_color, self.settings.bg_color)
        ---snip--
```

1 The **round()** function normally rounds a decimal number to a set number of decimal places given as the second argument. However, when you pass a negative number as the second argument , **round()** will round the value to the nearest 10, 100, 1000, and so on.
The code here tells Python to round the value of **stats.score** to the nearest 10 and store it in **rounded_score**.

2 a string formatting directive tells Python to insert commas into numbers when converting a numerical value to a string: for example, to output 1,000,000 instead of 1000000. Now when you run the game, you should see a neatly formatted, rounded score even when you rack up lots of points

HIGH SCORES

Every player wants to beat a game's high score, so let's track and report high scores to give players something to work toward. We'll store high scores in GameStats

game_stats.py

```
def __init__(self, vi_game):
        --snip--
        # High score should never be reset.
        self.high_score = 0
```

Because the high score should never be reset, we initialize high_score in **__init__()** rather than in **reset_stats().**

Next, we'll modify **Scoreboard** to display the high score. Let's start with the **__init__()** method

```
def __init__(self, vi_game):
        --snip--
       # Prepare the initial score images.
       self.prep_score()
       self.prep_high_score()
```

The high score will be displayed separately from the score, so we need a new method, **prep_high_score()**, to prepare the high score image

HIGHSCORES

Here’s the **prep_scor()** method

scoreboard.py

```
def prep_high_score(self):
      """Turn the high score into a rendered image."""
1     high_score = round(self.stats.high_score, -1)
      high_score_str = "{:,}".format(high_score)
2     self.high_score_image = self.font.render(high_score_str, True,
                                       self.text_color, self.settings.bg_color)
       # Center the high score at the top of the screen.
       self.high_score_rect = self.high_score_image.get_rect()
3      self.high_score_rect.centerx = self.screen_rect.centerx
4      self.high_score_rect.top = self.score_rect.top
```

1 We round the high score to the nearest 10 and format it with commas.

2 then generate an image from the high score

3 center the high **score_rect** horizontally

4 and set its top attribute to match the top of the score image

HIGHSCORES

The **show_score()** method now draws the current score at the top right and the high score at the top center of the screen:

scoreboard.py

```
def show_score(self):
    """Draw score to the screen."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
```

To check for **high_scores**, we'll write a new method, **check_high_score()**,in Scoreboard:

scoreboard.py

```
def check_high_score(self):
    """Check to see if there's a new high score."""
    if self.stats.score > self.stats.high_score:
        self.stats.high_score = self.stats.score
        self.prep_high_score()
```

The method **check_high_score()** checks the current score against the high score. If the current score is greater, we update the value of **high_score** and call **prep_high_score()** to update the high score's image.

HIGHSCORES

We need to call check_high_score() each time a virus is hit after updating the score in **_check_laser_beam_virus_collisions()**:

virus_invasion.py

```
def _check_laser_beam_virus_collisions(self):
    --snip--
    if collisions:
        for viruses in collisions.values():
            self.stats.score += self.settings.virus_points * len(viruses)
        self.sb.prep_score()
        self.sb.check_high_score()
    --snip-
```

We call **check_high_score()** when the collisions dictionary is present, and we do so after updating the score for all the viruses that have been hit.
The first time you play ***VirusInvasion*** *game*, your score will be the high score, so it will be displayed as the current score and the high score.
But when you start a second game, your high score should appear in the middle and your current score at the right

DISPLAY THE NUMBER OF ROBOTS

Finally, let’s display the number of robots the player has left, but this time, let’s use a graphic. To do so, we’ll draw robots in the upper-left corner of the screen to represent how many robots are left, just as many classic arcade games do First, we need to make Robot inherit from Sprite so we can create a group of robots:

robot.py

```
import pygame
from pygame.sprite import Sprite

1 class Robot(Sprite):
      """A class to manage the robot."""
      def __init__(self, vi_game):
          """Initialize the robot and set its starting position."""
2         super().__init__()
      --snip--
```

1 Here we import Sprite, make sure Robot inherits from Sprite

2 and call super() at the beginning of __init__().

DISPLAY THE NUMBER OF ROBOTS

Next, we need to modify Scoreboard to create a group of robots we can display. Here are the **import** statements for **Scoreboard**:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group
from robot import Robot
```

Because we're making a group of robots, we import the Group and Robot classes. Here's **__init__()**

scoreboard.py

```
def __init__(self, vi_game):
    """Initialize scorekeeping attributes."""
    self.vi_game = vi_game
    self.screen = vi_game.screen
    --snip--
    self.prep_score()
    self.prep_high_score()
    self.prep_robots()
```

We assign the game instance to an attribute, because we'll need it to create some robots. We call **prep_robots()**.

DISPLAY THE NUMBER OF ROBOTS

Here's **prep_robots():**

scoreboard.py

```
def prep_robots(self):
     """Show how many robots are left."""
1    self.robots = Group()
2    for robot_number in range(self.stats.robots_left):
         robot = Robot(self.vi_game)
3        robot.rect.x = 10 + robot_number * robot.rect.width
4        robot.rect.y = 10
5        self.robots.add(robot)
```

1 The **prep_robots()** method creates an empty group, **self.robots**, to hold the robot instances .

2 To fill this group, a loop runs once for every robot the player has left.

3 Inside the loop, we create a new robot and set each robot's x-coordinate value so the robots appear next to each other with a 10-pixel margin on the left side of the group of robots.

4 We set the y-coordinate value 10 pixels down from the top of the screen so the robots appear in the upper-left corner of the screen.

5 Then we add each new robot to the group robots.

DISPLAY THE NUMBER OF ROBOTS

Now we need to draw the robots to the screen:

scoreboard.py

```
def show_score(self):
    """Draw scores, level, and robots to the screen."""
     self.screen.blit(self.score_image, self.score_rect)
     self.screen.blit(self.high_score_image, self.high_score_rect)
     self.screen.blit(self.level_image, self.level_rect)
     self.robots.draw(self.screen)
```

To display the robots on the screen, we call **draw()** on the group, and Pygame draws each robot.
To show the player how many robots they have
we call **prep_robots()** when a robot is hit to update the display of robot images when the player loses a robot

virus_invasion.py

```
def _robot_hit(self):
    """Respond to robot being hit by virus."""
    if self.stats.robots_left > 0:
        # Decrement robots_left, and update scoreboard.
        self.stats.robots_left -= 1
        self.sb.prep_robots()
--snip--
```

We call **prep_robots()** after decreasing the value of **robots_left**, so the correct number of robots displays each time a robot is destroyed.

THANK YOU FOR YOUR LISTENING, AND WISH YOU GOOD LUCK