Space Invaders - Iterative Enhancement Plan (IEP)

For Person 1

Throughout, help teammates as needed. Use methods and instance variables per the UML class diagram. Delete temporary code when continuing to the next stage. **Coordinate with teammates whenever you touch the shared files: *main*, *Controller*, *Game* and *View*. Commit-and-push at each stage but beware of Git conflicts!**

0. [**Starting code**] A blank screen appears. The code is a bare-bones Model-View-Controller architecture.

1. [**Entire** team**,** with your instructor] A blank screen appears. The background, caption and screen size are set. There are files with stubs for all classes (no methods yet) and an instance of each object is constructed in Game's *__init__* method.

2. The ***Fighter*** class (in the *Fighter.py* file) has stubs for all its methods, per the UML class diagram. The constructor method (*__init__*) sets instance variables per parameters and (temporarily) prints a simple message.

3. The Fighter appears, centered horizontally and near the bottom of the screen, drawn from an image in the ***assets*** folder. Sub-stages:
   a. The Game's ***draw_game*** method calls the Fighter's ***draw*** method, which just *prints* "draw fighter", temporarily.
   b. The Fighter appears on the screen at a hard-coded place.
   c. The Fighter appears on the screen as specified.

4. The Fighter moves left/right 5 pixels per game loop cycle whenever the left/right arrow key is in the PRESSED state, but restricted so that the Fighter does not go more than ½ of the Fighter's width off the left or right edges of the screen. Sub-stages:
   a. Left/right keys just *print* "left", "right".
   b. Left/right keys make the Fighter move left/right.
   c. Fighter does not go more than ½ of its width off the screen.

Help Person 3 (Missile class implementor) complete their Stage 5 *while doing your own Stage 5*:

5. When the Space bar is pressed, the Fighter fires a Missile, which:
   a. Causes a "pew" sound (from the ***assets*** folder),
   b. Constructs a new Missile at the current position of the top of the Fighter, centered horizontally on the Fighter, and
   c. Adds that Missile to Missiles class (per the *add_missile* method in the Missiles class). This step also requires that the __init__ for the Fighter have the Missiles object as a parameter.

   Pressing the space bar repeatedly causes multiple Missile objects to appear at the Fighter (and move). *Holding* the Space bar down does NOT generate additional Missile objects.

[To be continued in Part 2]

After completing the above, help teammates complete their work.

Reminders:

1. ***Image / blit:***

```
self.image = pygame.image.load("../assets/fighter.png")
self.screen.blit(self.image, (self.x, self.y))
```

2. ***Draw line:***

```
pygame.draw.line(self.screen, self.color, (self.x, self.y),
    (self.x, self.y + self.height), self.width)
```

3. ***Play sound:***

```
self.fire_sound = pygame.mixer.Sound("../assets/pew.wav")
self.fire_sound.play()
```

4. ***Key interaction:***

```
if pressed_keys[pygame.K_LEFT]:  or
if key_was_pressed_on_this_cycle(pygame.K_SPACE, events):
```

Space Invaders - Iterative Enhancement Plan (IEP)

For Person 2

Throughout, help teammates as needed. Use methods and instance variables per the UML class diagram. Delete temporary code when continuing to the next stage. **Coordinate with teammates whenever you touch the shared files: *main*, *Controller*, *Game* and *View*. Commit-and-push at each stage but beware of Git conflicts!**

0. [**Starting code**] A blank screen appears. The code is a bare-bones Model-View-Controller architecture.

1. [**Entire** team**,** with your instructor] A blank screen appears. The background, caption and screen size are set. There are files with stubs for all classes (no methods yet) and an instance of each object is constructed in Game's *__init__* method.

2. The ***Enemy*** class (in the *Enemy.py* file) has stubs for all its methods, per the UML class diagram. The constructor method (*__init__*) sets instance variables per parameters and (temporarily) prints a simple message.

3. A single temporary Enemy appears on the screen as specified, drawn from an image in the ***Assets*** folder. Sub-stages:
   a. The Game's ***draw_game*** method calls the Enemy's ***draw*** method, which just *prints* "draw enemy", temporarily.
   b. The Enemy appears on the screen somewhere near the top of the screen.

4. At each cycle of the game loop, the Enemy moves sideways (to the right) per its horizontal speed, and when it gets 100 pixels from where it starts, it moves down per its vertical speed and reverses horizontal direction. Substages:
   a. Move sideways (to the right).
   b. Move down and reverse direction when 100 pixels from its start.

5. Enemies are drawn and move, per the Enemies class. Sub-stages:
   a. The **Enemies** class (note plural) has stubs for its methods, per the UM class diagram.
   b. The constructor method (*__init__*) sets instance variables per parameters and (temporarily) prints a simple message.
   c. A two-dimensional array of Enemies appears, centered horizontally, near the top of the screen, in 5 rows of 8 Enemy objects per row. The Enemies object constructs and stores the list of Enemy objects. The temporary Enemy is removed.
   d. Enemies move just like the temporary Enemy moved.

[To be continued in Part 2]

After completing the above, help teammates complete their work.

Reminders:

1. ***Image / blit:***

```
self.image = pygame.image.load("../assets/fighter.png")
self.screen.blit(self.image, (self.x, self.y))
```

2. ***Draw line:***

```
pygame.draw.line(self.screen, self.color, (self.x, self.y),
    (self.x, self.y + self.height), self.width)
```

3. ***Play sound:***

```
self.fire_sound = pygame.mixer.Sound("../assets/pew.wav")
self.fire_sound.play()
```

4. ***Key interaction:***

```
if pressed_keys[pygame.K_LEFT]:  or
if key_was_pressed_on_this_cycle(pygame.K_SPACE, events):
```

Space Invaders - Iterative Enhancement Plan (IEP)

For Person 3

Throughout, help teammates as needed. Use methods and instance variables per the UML class diagram. Delete temporary code when continuing to the next stage. **Coordinate with teammates whenever you touch the shared files: *main*, *Controller*, *Game* and *View*. Commit-and-push at each stage but beware of Git conflicts!**

0. [**Starting code**] A blank screen appears. The code is a bare-bones Model-View-Controller architecture.

1. [**Entire** team**,** with your instructor] A blank screen appears. The background, caption and screen size are set. There are files with stubs for all classes (no methods yet) and an instance of each object is constructed in Game's *__init__* method.

2. The ***Missile*** class (in the *Missile.py* file) has stubs for all its methods, per the UML class diagram. The constructor method (*__init__*) sets instance variables per parameters and (temporarily) prints a simple message.

3. A single temporary Missile appears on the screen as specified, drawn from an image in the ***Assets*** folder. Sub-stages:

   e. The Game's ***draw_game*** method calls the Missile's ***draw*** method, which just *prints* "draw missile", temporarily.

   f. The Missile appears on the screen somewhere near the bottom of the screen.

4. At each cycle of the game loop, the Missile moves up per its speed.

Help Person 1 (Fighter class implementor) complete their Stage 5 *while doing your own Stage 5*:

5. Missiles are drawn and move, per the ***Missiles*** class (not the plural). Sub-stages:

   a. The Missiles class has stubs for its methods, per the UM class diagram.

   b. Two temporary Missile objects are in the *missiles_list*, at different places on the screen. Each Missile in the *missiles_list* is drawn and moves just like the temporary Missile moved.

   c. The *add_missile* method is implemented. The temporary Missile objects are removed. [Now test using the Fighter's *fire* method.]

[To be continued in Part 2]

After completing the above, help teammates complete their work.

Reminders:

1. Image / blit:

```
self.image = pygame.image.load("../assets/fighter.png")
self.screen.blit(self.image, (self.x, self.y))
```

2. Draw line:

```
pygame.draw.line(self.screen, self.color, (self.x, self.y),
    (self.x, self.y + self.height), self.width)
```

3. Play sound:

```
self.fire_sound = pygame.mixer.Sound("../assets/pew.wav")
self.fire_sound.play()
```

4. Key interaction:

```
if pressed_keys[pygame.K_LEFT]:  or
if key_was_pressed_on_this_cycle(pygame.K_SPACE, events):
```