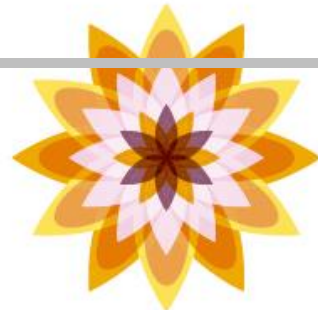
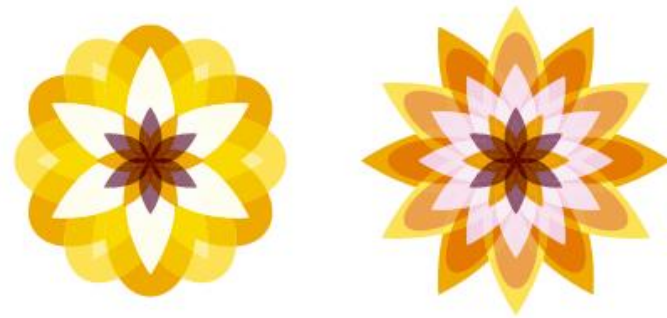


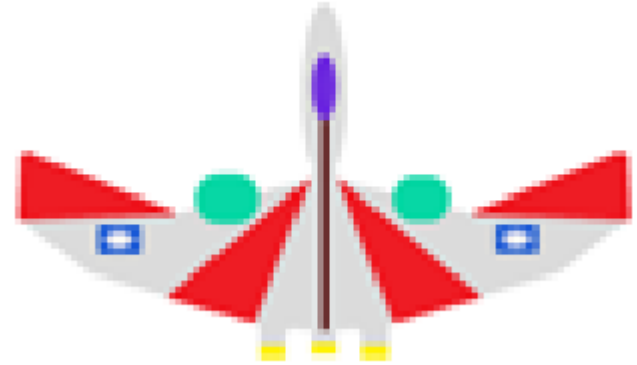
*Chapter 09*

# 미사일을 쏘는 파이터



# 1. 파이터 클래스

- 파이터 fighter 클래스를 후다닥 만듭시다.
- 언제나 그렇듯이 이미지가 필요합니다.
- Images 폴더에 넣는 것만 잊지 마세요.
- 흰색 배경이므로 투명한 배경을 만들기 위해 컬러키를 흰색으로 정합니다.
- 아래 코드는 셋업 아무 데나 넣으면 됩니다.



```
fighter_image = pygame.image.load("images/fighter.png").convert()  
fighter_image.set_colorkey((255,255,255))
```

# 1. 파이터 클래스

- 지금부터 본격적으로 파이터 클래스를 만들어 봅시다.
- 악당 클래스 밑에 아래 코드를 추가하세요.

```
class Fighter:
    def __init__(self):
        self.x = 320
    def move(self):
        if pressed_keys[K_LEFT] and self.x > 0:
            self.x -= 3
        if pressed_keys[K_RIGHT] and self.x < 540:
            self.x += 3
    def draw(self):
        screen.blit(fighter_image, (self.x, 591))
```

# 1. 파이터 클래스

- 파이터의 x좌표는 일단 320으로 정합니다.
- y좌표는 항상 591로 고정할 것이므로 draw() 함수 안에 591이라고 (변수가 아닌) 숫자로 씁니다.
- 591인 이유는 스크린의 높이는 650픽셀이고, 파이터의 높이는 59픽셀이므로, 파이터가 스크린 아래쪽에 나타나게 하기 위해서입니다.
- 파이터 클래스의 move() 함수는 파이터를 왼쪽 또는 오른쪽으로 움직입니다.
- Move() 함수 안의 첫 번째 if 문을 보면, 두 가지 조건이 먼저 충족돼야 다음 줄이 실행되는 것을 알 수 있습니다.
- 왼쪽 화살표 키가 눌러야 하고, self.x가 ()보다 커야 합니다.
- 파이터가 스크린 밖으로 떨어지는 것을 막기 위해서입니다.
- 만약 self.x가 ()보다 같거나 작아지면, 두 번째 조건은 거짓을 돌려주고 파이터는 왼쪽으로 더 이상 가지 않을 것입니다.

# 1. 파이터 클래스

- 두 번째 if 문도 같은 식입니다. 제한은 540입니다.
- 파이터의 오른쪽 날개 끝이 스크린의 오른쪽 끝에 닿을 때의 파이터의 x좌표는 540이기 때문입니다.
- 파이터는 너비가 100픽셀이니깐요.
- 키가 눌렸는지 확인하기 위한 눌린 키들 리스트 `pressed_keys`도 만듭니다.

```
while 1:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
    pressed_keys = pygame.key.get_pressed()
```

# 1. 파이터 클래스

- 이제 파이터의 인스턴스를 만듭시다.
- 파이터 클래스 바로 밑에 씁니다. 들여 쓰기는 하지 않습니다.

```
fighter = Fighter()
```

- 마지막으로 while 1: 루프 안에서 파이터의 move() 함수와 draw() 함수를 부릅니다.

```
while 1:  
    (중략)  
    screen.fill((0, 0, 0))  
    fighter.move()  
    fighter.draw()
```

# 1. 파이터 클래스


- `screen.fill()` 함수가 실행된 후, `draw()` 함수를 불러와야 합니다. 그렇지 않으면 스크린이 파이터를 덮습니다.
- 두 함수는 같이 쓰는 것이 좋습니다. 이제 파이터는 끝났습니다.

## 2. 미사일 공장

- 미사일은 빗방울과 거의 비슷합니다.
- 왼쪽이나 오른쪽으로 움직이는 객체로부터 발사되고, 수직으로 움직이고, 스크린 밖으로 나가면 삭제돼야 합니다.
- 미사일 클래스를 봅시다.
- 파이터 클래스 밑에 만드는 걸 추천하지만 사실 클래스 사이에 아무 데나 만들어도 됩니다.



## 2. 미사일 공장

```
class Missile:
    def __init__(self,x):
        self.x = x
        self.y = 591
    def move(self):
        self.y -= 5
    def off_screen(self):
        return self.y < -8
    def draw(self):
        pygame.draw.line(screen,(255,0,0),(self.x,self.y),
 (self.x,self.y+8),1)
```

## 2. 미사일 공장

- 미사일 클래스는 빗방울 클래스와 비슷하지만, 미사일 클래스의 `__init__()` 함수는 `self` 외에 오직 1개의 인자, `x`만을 가집니다.
- 미사일 발사 장소를 결정하는 `x`값은 파이터의 좌우 움직임에 따라 바뀝니다.
- 파이터가 위아래로 움직이지 않으므로 미사일 `y`좌표의 초기값은 항상 같습니다.
- `Move()` 함수는 미사일을 스크린 위쪽으로 쏘아 올립니다.
- `off_screen()` 함수는 미사일이 스크린 위쪽으로 나가면 참값을 돌려줍니다.

## 2. 미사일 공장

- 이제 미사일 리스트를 만듭니다.
- 미사일을 많이 만들어야 하므로 리스트가 꼭 필요합니다.

```
badguys = []  
fighter = Fighter()  
missiles = []
```

- 미사일 리스트를 만든 뒤에는 while 루프가 필요합니다.
- 미사일 함수들을 불러 오고 미사일을 삭제해야 하니까요.

## 2. 미사일 공장

- 아래 코드는 악당들을 제어 하는 루프 밑에 씁니다.

```
while 1:
    (중략)
    i = 0
    while i < len(badguys):
        (중략)
        i += 1
    i = 0
    while i < len(missiles):
        missiles[i].move()
        missiles[i].draw()
        if missiles[i].off_screen():
            del missiles[i]
        i -= 1
    i += 1
    pygame.display.update()
```

## 2. 미사일 공장

- 악당 제어용 while 루프와 매우 비슷하죠?
- 두 루프 다 i를 사용해도 괜찮습니다.
- 한 루프 안에서 i를 만들어 쓰고 나면, 다음 루프에서는 0으로 리 셋 후 사용하니까요.
- 서로 간섭하지 않아요.

### 3. 미사일 발사

- 미사일과 빗방울, 악당의 차이는 무엇일까요?
- 바로 미사일을 만드는 방법이 다르다는 것입니다.
- 우리는 미사일 발사를 위해 키 key를 사용할 것입니다.
- 키보드의 특정 키를 누르면 미사일이 발사되게 말이죠.
- 그러려면 파이터 클래스 안에 미사일을 만들고, 미사일 리스트에 추가하는 함수를 써야 합니다.
- 게임 루프 안에 발사키가 눌렸을 때 관련 함수를 불러오는 코드도 써야 합니다.

### 3. 미사일 발사

- 다음 함수를 파이터 클래스 안에 추가합니다.

```
class Fighter:  
(중략)  
    def fire(self):  
        missiles.append(Missile(self.x+50))
```

- 파이터로부터 발사될 때 미사일 위치는 파이터 위치와 같습니다.
- Fire() 함수를 파이터 클래스 안에 썼으니깐요.
- Fire() 함수는 미사일 리스트에 하나의 미사일을 추가합니다.
- 구름 클래스의 rain() 함수와 거의 같지요.
- 위 코드의 2번째 줄을 봅시다. Missiles는 미사일 클래스라는 말입니다.
- 미사일 클래스의 \_\_init\_\_() 함수 안의 "self" 인자와 관련있지요.

### 3. 미사일 발사

- `Missile(self.x+50)`은 미사일 `__init__()` 함수 안의 “x”인자와 관련 있지요.
- 파이터의 x좌표인 `self.x`에 50을 더한 것이지요.
- 이 코드는 `rain()` 함수보다 조금 더 간단합니다.
- 미사일은 임의의 위치에서 발사될 필요가 없으니까요.
- 항상 파이터의 중심에서 발사되지요.
- 좌표는 파이터의 x좌표에 50픽셀을 더한 값입니다. 파이터는 너비가 100픽셀이 거든요.
- 왜 인자 사이에 쉼표가 없는지 궁금한가요?
- `Missile()`은 미사일 생성자입니다.
- `self.x+50`은 미사일 생성자의 인자입니다.
- 미사일 생성자는 `append()` 함수의 인자입니다.
- 새로운 객체를 만들 때는 클래스의 `__init__()` 함수가 요구하는 인자를 갖고 있는 생성자를 사용합니다.



### 3. 미사일 발사

- Fire() 함수를 미사일 클래스에 넣어야 한다고 생각할 수도 있습니다.
- 결국 발사되는 건 미사일이니깐요.
- 그렇게 할 수 없는 이유는 미사일이 발사되기 전에는 함수를 호출할 미사일 자체가 존재하지 않기 때문입니다.

## 4. 발사 버튼

- for 루프 안에 음영 표시된 코드를 추가하세요.

```
for event in pygame.event.get():  
    if event.type == QUIT:  
        sys.exit()  
    if event.type == KEYDOWN and event.key == K_SPACE:  
        fighter.fire()  
    pressed_keys = pygame.key.get_pressed()
```

- 앞에서 for 루프로 게임 끝내기를 설명했습니다.
- 그때는 입 아이콘을 마우스 클릭하는 이벤트가 있는지 확인했지요.
- 프로그래밍에서는 마우스 클릭도, 키보드 눌림도 이벤트입니다.

## 4. 발사 버튼

- 4번째 줄 if문은 키다운KEYDOWN 이벤트를 확인합니다.
- 키다운은 키보드가 눌림 상태로 바뀌었는지 확인하는 것 입니다.
- 이미 눌린 상태라면 키다운으로 감지할 수 없습니다.
- 키업 KEYUP은 키다운의 반대입니다.
- `for event in pygame.event.get():`는 키보드 눌림, 마우스 클릭 같은 이벤트 확인 후 이벤트의 리스트를 만듭니다.
- 4번째 줄 if 문의 첫 번째 조건은 `event.type == KEYDOWN`입니다.
- 리스트 안 이벤트 중에 키다운 이벤트가 있는지 물어보는 것입니다.
- 두 번째 조건은 `event.key == K _ SPACE`입니다.
- 만약 눌린 키가 있다면 그 중에 스페이스키가 있는지를 묻습니다.
- 첫 번째 조건은 생략할 수 없습니다. 생략하면 충돌이 일어나니까요.

## 4. 발사 버튼

- 두 번째 조건을 생략하면 스페이스키가 아닌 아무 키나 눌러도 참이 됩니다.
- if 문의 두 조건이 모두 참이면, `fighter.fire()` 함수가 불려와 미사일이 만들어집니다.
- 계속 스페이스키를 누르고 있어도 미사일은 오직 하나만 만들어집니다.
- 왜냐하면 키다운 이벤트, 즉 키를 누르는 일은 1번만 일어나기 때문입니다.

## 5. 미사일 이미지 사용

- 많은 게임에서 미사일을 그냥 짧은 선으로 표시하지만, 이미지를 사용할 수도 있습니다. 어떻게 하는지는 이미 알고 있지요?
- 먼저 이미지가 필요합니다.
- 웹사이트에서 missile.png 파일을 다운받으세요.
- 너비가 몇 픽셀밖에 안 되는 이미지는 디테일하게 만들기 어렵습니다.
- 프로그램의 셋업에 아래 코드를 추가합니다.

```
missile_image = pygame.image.load("images/missile.png").convert()  
missile_image.set_colorkey((255,255,255))
```

## 5. 미사일 이미지 사용

- 미사일 클래스 안의 draw() 함수를 다음과 같이 바꿉니다.

```
class Missile:  
(중략)  
    def draw(self):  
        pygame.draw.line(screen,(255,0,0),(self.x,self.y),(self.x,self.y+8),1)  
        screen.blit(missile_image,(self.x,self.y))
```

- 이 미사일의 너비는 8픽셀입니다.
- 이렇게 쓰면 미사일 왼쪽 끝이 파이터의 중심에 맞춰집니다.

## 5. 미사일 이미지 사용

- 제대로 정렬하려면 아래같이 고쳐서 미사일의 x좌표를 미사일 너비의 반만큼 왼쪽으로 옮깁니다.

```
def draw(self):  
    screen.blit(missile_image, (self.x-4, self.y))
```



**Thank You**

---