Chapter 16

임의로 바꾸기

1. d, dx, dy값 업데이트

- 지금은 공이 항상 똑같은 각도로 튕겨 나갑니다.
- 공이 어디로 이동할지 예측하기 쉽지요.
- 공 클래스에서 설명한 bounce() 함수 기억나나요?
- 공이 스크린의 가장 위나 아래에 닿았는지 감지해서 만약 닿았다면 dy의 부호를 바꿉니다.
- 공이 배트에 닿으면 for 루프에서는 dx의 부호를 바꿨고요.
- 튕겨 나가는 각도를 임의로 바꾸려면, d의 값을 바꿔야 합니다.
- 처음에는 각 d에서 공을 발사했지만 공이 튕겨 나가고 나면 더이상 d 방향이 아닙니다.
- 공은 진행 각을 바꾸었지만 d의 값은 변화가 없습니다.
- 공이 배트에 맞고 튕겨 나가는 각도를 임의로 바꾸려면, 공이 배트를 친 각도를 정확히 알아야 합니다.

1. d, dx, dy값 업데이트

- 다행히 우리에게는 dx, dy의 값을 받아와서 각도 d를 알려 주는 math.atan()라는 함수가 있습니다.
- 따라서 앞으로 다시 가서 d의 새로운 값을 적용해 봅시다.
- bounce() 함수에 다음과 같이 추가합니다.

```
class Ball:
(중략)
    def bounce(self):
        if self.y<= 0 or self.y >= 550:
            self.dy *= -1
            self.d = math.atan2(self.dx,self.dy)
        for bat in bats:
```

1. d, dx, dy값 업데이트

- d가 아니라 self.d라고 쓴 것을 눈치챘나요?
- 지금까지 d는 __init__() 함수 안에서만 사용했습니다.
- 이제 bounce() 함수 안에서 사용했으니 다시 앞으로 가서 __init__() 함수 안에 있는 모든 d를 self.d로 바꿔야 합니다.

```
class Ball:
    def __init__(self):
        self.d=(math.pi/3)*random.random()+(math.pi/3)+math.pi*random.randint(0,1)
        self.dx=math.sin(self.d)*12
        self.dy=math.cos(self.d)*12
        self.x=475
        self.y=275
```

- 공이 스크린 가장 위나 가장 아래에서 튕겨 나올 때, dy의 부호는 바뀌고 우리는 d를 다시 계산해야 하는 거지요.

1. d, dx, dy값 업데이트

- 이제 정확한 d 값을 알았으므로, 공이 배트에 맞고 튕겨 나오는 방식을 바꿀 수 있습니다.
- 각에 -1을 곱하는 것은 dx에 -1을 곱하는 것과 같습니다.
- 원의 중심에서 멀어질 때 각에 -1을 곱하는 것은, 수직의 벽이나 배트에 부딪혀 튕겨 나오는 것과 같습니다.
- 따라서 정확한 각도를 아는 것이 중요합니다.
- bounce() 함수 안에 있는 for 루프에서 dx를 d로 바꿉니다.

```
def bounce(self):
(중략)
    for bat in bats:
        if pygame.Rect(bat.x,bat.y,6,80).colliderect(self.x,self.y,50,50) and abs(self.
dx)/self.dx == bat.side:
            self.dx d *= -1
```

1. d, dx, dy값 업데이트

- self.d를 바꿨으므로 dx와 dy도 업데이트해야 합니다.
- 만약 바꾸지 않으면 self.d가 변했는데도 예전 값 그대로일 것입니다.
- 그러면 공은 튕기지 않고 계속 가겠지요.
- 따라서 for 루프의 가장 아래에 코드 두 줄을 추가합니다.
- 이제 전체 bounce() 함수 완성본은 다음과 같습니다.

```
def bounce(self):
    if self.y<=0 or self.y>=550:
        self.dy *=-1
        self.d = math.atan2(self.dx,self.dy)
    for bat in bats:
        if pygame.Rect(bat.x,bat.y,6,80).colliderect(self.x,self.y,50,50) and abs(self.
dx)/self.dx == bat.side:
            self.d *= -1
            self.dx=math.sin(self.d)*12
            self.dy=math.cos(self.d)*12
```

2. 임의의 각

- 이제 self.d를 임의의 값으로 바꿔서 배트에 맞고 튕겨 나가는 방향을 예측하기 어렵게 해 봅시다.

```
for bat in bats:
    if (pygame.Rect(bat.x,bat.y,6,80).colliderect(self.x,self.y,50,50) and abs(self.
dx)/self.dx == bat.side):
        self.d += random.random()*math.pi/4 - math.pi/8
        self.d *= -1
        self.dx=math.sin(self.d)*12
        self.dy=math.cos(self.d)*12
```

실행

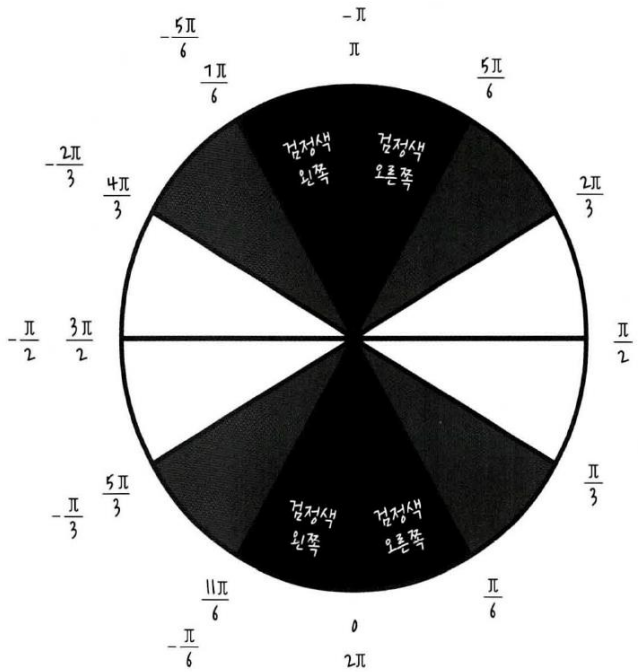
- 추가된 부분은 이전에 만든 유사 코드와 같은 방식으로 작동합니다.
- random.random()*math.pi/4는 0에서 π/4 사이의 임의의 각을 만듭니다.
- 거기에 π/8을 빼어 –π/8에서 π/8 (-22.5°에서 22.5°) 사이의 각을 만듭니다.
- 이 각을 self.d에 더해 각의 크기를 조금 커지거나 작아지게 바꿉니다.

2. 임의의 각

- 각도를 임의로 바꾸는 것은 좋지만, 문제가 생길 수 있습니다.
- 프로그래밍하다 보면 종종 생기는 일이지요.
- 뭔가 추가하면 의도하지 않았던 여기저기서 문제가 발생합니다.
- 여기서의 문제는 각이 거의 수직에 가까워져 공이 스크린 위아래로만 움직이고, 옆으로는 거의 움직이지 않을 수도 있다는 것입니다.
- 따라서 각의 크기에 조건을 걸어야 합니다.

3. 수직으로 튕겨 나가면 안돼!!

- 다음 마차 바퀴 그림을 보세요. 공은 흰색 구역 어딘가로 발사됩니다.
- self.d를 임의의 값으로 바꾸면 아마 회색이나 검은색 구역으로 움직이겠지요.
- 회색은 괜찮습니다.
- 이 구역에서는 적당한 속도로 스크린 위를 지그재그로 움직입니다.
- 하지만 self.d가 검은색 구역으로 들어가면 문제가 생깁니다.
- 따라서 검은색 구역으로 들어가면 흰색 구역으로 돌아가라고 명령하는 코드를 써야 합니다.



3. 수직으로 튕겨 나가면 안돼!!

- 다음 코드를 추가해 공의 움직임 각도를 제한합니다.

```
for bat in bats:
    if (pygame.Rect(bat.x,bat.y,6,80).colliderect(self.x,self.y,50,50) and abs(self.
dx)/self.dx == bat.side):
        self.d += random.random()*math.pi/4 - math.pi/8
        if (0 < self.d < math.pi/6) or (math.pi*5/6 < self.d < math.pi):
            self.d = ((math.pi/3)*random.random() + (math.pi/3))
        elif (math.pi < self.d < math.pi*7/6) or (math.pi*11/6 < self.d < math.pi*2):
            self.d = ((math.pi/3)*random.random()+(math.pi/3))+math.pi
        self.d *= -1
        self.d %= math.pi*2

        self.dx = math.sin(self.d)*12
        self.dy = math.cos(self.d)*12
```

3. 수직으로 튕겨 나가면 안돼!!

- 꽤 복잡해 보이지만 이해하면 어렵지 않습니다.
- 다음은 위 코드의 의사 코드입니다.

```
공을 맞힐 수 있는 모든 배트에 대해
    공이 배트에 맞고 배트 앞으로 나가면
        공이 움직이는 각(d)을 조금 바꿔라
        만약 새 각이 너무 오른쪽으로 서 있으면
            d를 오른쪽으로 조금 돌려라
        또는 각이 왼쪽으로 너무 서 있으면
            d를 왼쪽으로 조금 돌려라
        d에 -1을 곱해라
        각의 크기를 양수로 만들어라

        dx의 값에 맞춰 다시 계산하라
        dy의 값에 맞춰 다시 계산하라
```

4. 모드: 4π/3은 -2π/3과 같아

- 추가 부분의 첫째 줄에서는 self.d가 0과 π/6 사이인지, 5π/6과 π 사이인지 확인합니다.
- 앞의 그림의 "오른쪽 검은색" 구역인지를 확인하는 것입니다.
- 만약 그렇다면 다음 줄에서 self.d를 오른쪽 흰색 구역 어디쯤으로 바꿉니다.
- 이 코드는 원래 공 발사 코드에서 가져온 것이지만 끝에 pi*random.random(0, 1)이 없습니다.
- 왜냐하면 공을 오른쪽으로만 움직이게 하고 싶으니까요.
- elif로 시작하는 코드는 공이 '왼쪽 검은색' 구역에 있는지 확인하고 '왼쪽 흰색' 구역으로 보냅니다.
- 음영 표시된 코드의 4번째 줄 끝에서 pi를 더하는 이유는, 그렇게 해야 공이 왼쪽으로 가기 때문입니다.
- 180°를 더하는 것과 같지요.

4. 모드: 4π/3은 -2π/3과 같아

- 아직도 문제가 남았습니다.
- 14장의 라디안 그림에서 10시 방향인 각은 4π/3 또는 -2π/3입니다.
- 같은 각도지요.
- 공이 처음 발사될 때는 첫 번째처럼 4π/3라고 표시하지만, 공이 2π/3 방향으로 가다가 오른쪽 배트에 맞아 튕겼을 때는 각에 -1을 곱하므로 -2π/3라고 표시합니다.
- 우리가 방금 쓴 코드는 4π/3이라고 썼을 때만 감지할 수 있습니다.
- 마차 바퀴의 왼쪽은 양수만 사용하고, 음수는 사용하지 않기 때문입니다.
- 음수를 쓰면 혼란이 생깁니다.
- 고로 다음 코드를 추가했던 것입니다.

4. 모드: 4π/3은 -2π/3과 같아


```
self.d *= -1
self.d %= math.pi*2
```

- 앞에서 본 적 있죠?
- 이 코드는 음의 값을 그에 맞는 양의 값으로 바꾸어 줍니다.
- % 표시는 모듈러(modulus) 연산을 의미합니다.
- 모듈러는 어떤 각이든지 0부터 해당하는 양수까지의 값으로 바꾸어 표시합니다.
- 음수인 각을 넣으면 그에 해당하는 양수로 바꾸어 준다는 말입니다.
- 각이 2π(360°)보다 크면, 0에서 다시 시작합니다.
  - 위 코드의 math.pi*2 부분입니다.
- 따라서 3π를 모듈러 연산에 넣으면 π가 나옵니다.
  - -3π의 모듈러 연산의 결과도 π입니다.
- 이제 임의로 각을 바꾸어도 게임을 망치지 않도록 고쳤습니다.

5. 스피드의 필요성

- 두 번째로 고칠 것은, 공이 배트와 부딪힐 때마다 가속되도록 하는 것입니다.
- 스피드(speed) 변수를 공 클래스의 __init__()함수 안에 추가합시다.

```
class Ball:
    def __init__(self):
        self.d = ((math.pi/3)*random.random()+(math.pi/3))+math.pi*random.randint(0,1)
        self.speed = 12
        self.dx = math.sin(self.d)*12 self.speed
        self.dy = math.cos(self.d)*12 self.speed
        self.x = 475
        self.y = 275
```

실행

5. 스피드의 필요성

- bounce() 함수에 아래 음영 표시된 코드를 추가합니다.

```
def bounce(self):
    if (self.y <= 0  and self.dy<0) or (self.y >= 550 and self.dy >0):
        self.dy *= -1
        self.d = math.atan2(self.dx,self.dy)
    for bat in bats:
        if (pygame.Rect(bat.x,bat.y,6,80).colliderect(self.x,self.y,50,50) and
abs(self.dx)/self.dx == bat.side):
        self.d += random.random()*math.pi/4 -math.pi/8
        if (0 < self.d < math.pi/6) or (math.pi*5/6 < self.d < math.pi):
            self.d = ((math.pi/3)*random.random()+(math.pi/3))
        elif (math.pi < self.d < math.pi*7/6) or (math.pi*11/6 < self.d < math.pi*2):
            self.d = ((math.pi/3)*random.random()+(math.pi/3))+math.pi
        self.d *= -1
        self.d %= math.pi*2

        if self.speed < 20:
            self.speed *= 1.1
        self.dx = math.sin(self.d)* 12 self.speed
        self.dy = math.cos(self.d)* 12 self.speed
```

실행

6. 공을 쳐 보자

- 공이 배트에 닿을 때마다 자동으로가 아니라 플레이어들이 공을 배트로 때릴 때만 속도가 빨라지게 해 봅시다.
- 배트 관련 코드부터 만들어 봅시다.
- 왼쪽 플레이어가 'q'를 누르거나, 오른쪽 플레이어가 오른쪽 'Shift'를 눌렀을 때, 배트가 5/100초 동안 스크린의 중심 쪽으로 10픽셀 정도 점프한 후, 다시 제자리로 돌아오도록요.
- 먼저 배트 클래스에 마지막으로 공을 친 시각을 기록할 수 있는 변수를 만듭니다
- 이름은 마지막 공친 시각 (lastbop)이라고 합시다.
- 이 변수는 클래스 밖에서 쓸 일이 전혀 없기 때문에 배트 클래스 안에 만듭니다.

6. 공을 쳐 보자

- 배트 클래스 안에 bop() 함수도 만듭니다.

```
class Bat:
    def __init__(self,ctrls,x,side):
        self.ctrls=ctrls
        self.x=x
        self.y=260
        self.side=side
        self.lastbop = 0
(중략)
    def bop(self):
        if time.time() > self.lastbop + 0.3:
            self.lastbop = time.time()
```

- bop() 함수는 마지막으로 공을 친 시각을 기록하고, 최소 0.3초마다 마지막 공친 시각 (lastbop) 변수를 업데이트하지 못하도록 합니다.
- 플레이어가 쉬지 않고 계속 공을 치지 못하도록 말입니다.

6. 공을 쳐 보자

- 그러면 너무 쉬워지지요.
- 프로그램 첫 줄에 time 모듈을 가져오는 것도 잊지 마세요.

```
import pygame, sys, math, random, time
```

- 게임이 처음 시작될 때 마지막 공친시각은 0입니다.
- bop() 함수가 불려오면, if 문은 참값을 돌려주고, 마지막 공친시각은 현재 시각으로 설정됩니다.
- 만약 0.3초가 지나지 않았는데도 함수가 불려지면 if 문은 거짓이라고 대답하고 마지막 공친시각은 업데이트되지 않습니다.
- 0.3초가 지나면 if 문은 참이라고 대답하고 마지막 공친시각은 새로운 값을 갖습니다.

7. 마지막 공친 시각 변수 사용방법

- 이제 마지막 공친시각 변수를 어떻게 사용하는지 알아봅시다.
- offset이라는 변수를 배트 클래스 (draw()) 함수 안에 넣습니다.
- offset은 무슨 일을 할까요?

```
def draw(self):
    offset = -self.side*(time.time() < self.lastbop+0.05)*10
    pygame.draw.line(screen,(255,255,255),(self.x+offset,self.y),(self.x+offset,self.y
+80),6)
```

- 오른쪽 플레이어가 공을 치면 오른쪽 배트가 10픽셀만큼 왼쪽으로 이동해야 합니다.
- 마찬가지로 왼쪽 플레이어가 공을 치면 왼쪽 배트가 10픽셀만큼 오른쪽으로 이동해야 합니다.

7. 마지막 공친 시각 변수 사용방법

- 어떤 키가 눌렸는지에 따라 오른쪽 또는 왼쪽 배트에 대해 bop() 함수가 실행됩니다.
- 어떤 키를 누를 때 실행되게 할지는 정하기 나름입니다.
- 왼쪽 플레이어는 'q'를 눌렀을 때, 오른쪽 플레이어는 오른쪽 'Shift'를 눌렀을 때로 정하겠습니다.
- offset이 만들어지는 2번째 줄을 보면 이 수식의 값은 세 수식의 곱으로 이루어져 있습니다.
- 첫 번째 변수는 -self.side입니다.
- 왼쪽 배트의 경우 -1이고, 오른쪽 배트의 경우 1입니다.
- 지금껏 왼쪽 배트에는 1을 (1은 -1에 -1을 곱한 거죠), 오른쪽 배트에는 -1을 곱했습니다.
- 공을 칠 때 배트가 어느 쪽으로 움직일지 알 수 있겠지요?

7. 마지막 공친 시각 변수 사용방법

- 마지막 수식은 10입니다.
- 이 수식은 배트가 얼마나 멀리 이동할지 알려 줍니다.
- 가운데 수식 (time.time() < self.lastbop + 0.05)는 조금 이상해 보입니다.
- 이 코드는 참이나 거짓 값을 돌려 줍니다.
- 키가 눌리면, bop() 함수가 불려오고 마지막 공친시간 (lastbop)이 time.time() 으로 정해집니다.
- 그 후 0.05초간은 참값을 돌려주지만 이후에는 거짓값을 돌려줍니다.
- 왼쪽 배트가 함수를 부르면 0.05초간 offset은 1x1x10입니다.
- 오른쪽 배트가 부르면, 0.05초간 offset의 값은 -lx1x10이 되겠지요.
- 0.05초가 지나면 가운데 수식의 값은 0이 됩니다.
- 양쪽 모두에 대해 offset은 0이 됩니다.

8. 공을 치면 속도가 빨라져

- 공을 치도록 해 봅시다.
- 앞에서 배트와 공 사이의 충돌이 감지되면, 속도가 20이 될 때까지 1.1을 곱하는 코드를 bounce() 함수 안의 for 루프에 썼습니다.
- 그 코드를 다음과 같이 바꿉니다 .

```
if self.speed < 20:
    self.speed *= 1.1
if time.time() < bat.lastbop + 0.05:
    self.speed *= 1.5
```

- 이 코드는 지난 0.05초 사이에 bop() 함수가 불려오면, 속도에 1.5를 곱하라는 것입니다.
- 이 속도에 맞춰 플레이하려면 약간 연습이 필요하겠지만, 0.05초가 적당한 것 같습니다.
- 0.05를 0.1로 바꾸면 공을 치기가 훨씬 쉬워질 것입니다.

8. 공을 치면 속도가 빨라져

- 위 코드의 1번째 줄을 다음과 같이 바꾸어 속도 제한을 20으로 걸어 봅시다.

```
if time.time() < bat.lastbop + 0.05 and self.speed < 20:
    self.speed *= 1.5
```

- 파이썬은 참을 1, 거짓을 0이라고 생각한다고 설명했습니다.
- 파이썬 쉘을 불러와서 2개의 참인 문장을 그냥 더해 보세요
- (2<3) + (8>4)라고 쓰면 파이썬은 2라는 답을 돌려줍니다
- (True+ True+ True)**2 같은 것도 해 보세요
- 또는 False*82같은 것도 해 보세요
- 또는 (2 > 9)*10 같은 것도 해 보세요
- True과 False을 쓸 때는 첫 문자는 대문자여야 합니다

8. 공을 치면 속도가 빨라져

- 마침내 bop() 함수를 불러옵니다.
- 앞에서 말했듯이, 'q' 또는 오른쪽 'Shift'를 누르는 것을 공 치는 것으로 정합니다
- 이것은 끝내기 섹션이 들어 있는 for 루프 안에서 일어납니다.
- 미사일 발사 버튼을 만들었던 것과 같은 방식입니다.

```
for event in pygame.event.get():
    if event.type == QUIT:
        sys.exit()
    if event.type == KEYDOWN:
        if event.key == K_q:
            bats[0].bop()
        if event.key == K_RSHIFT:
            bats[1].bop()
pressed_keys = pygame.key.get_pressed()
```

9. 버그 미리 수정: 달라붙지 마!!

- 또 다른 버그를 이야기해 볼게요.
- 공은 배트에 들러붙기도 하지만 스크린의 가장 위 또는 아래에 붙기도 합니다.
- 이 일은 공이 배트에 맞고 튕겨 그대로 스크린의 가장 위나 아래에 닿을 때 일어납니다.
- 이 프로그램은 공이 위나 아래에 닿았는지 감지하지만 만약 임의로 튕겨서 self.d 값이 감소되면, 다음 루프에서 공은 스크린의 위나 아래에서 완벽히 떨어져 나오지 않을 것입니다.
- 그러면 계속 다시 시도하고 튕기겠지요.
- 이것을 고치기 위해서는 공이 방금 부딪힌 모서리(위나 아래)를 향하여 움직이고 있을 때만 튕기도록 해야 합니다.
- 그래서 앞에서 bounce() 함수 안에 이렇게 썼답니다.

9. 버그 미리 수정: 달라붙지 마!!

- 그래서 앞에서 bounce() 함수 안에 이렇게 썼답니다.

```
if (self.y <= 0 and self.dy < 0) or (self.y >= 550 and self.dy > 0):
```

- 이러면 공은 스크린의 가장 위에 닿고 나서 위쪽으로 향하거나(dy < 0), 스크린의 가장 아래에 닿고 아래로 향할 때 (dy > 0)만 튕기게 됩니다.
- 평소대로 프로그램을 만들었고, 버그도 고쳤습니다.
- 처음 쓴 코드는 제대로 작동하지 않는 경우가 대부분입니다.
- 프로그래밍을 잘하게 될 수록 더 그렇습니다.

Thank You