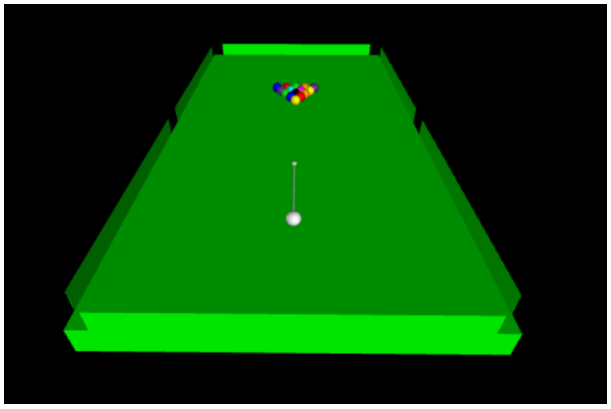
일반 물리 및 시뮬레이션 과제5

22011831 김형규

1. 게임 설명
2. 코드 설명

1. 게임 설명



흰 공을 이용하여 다른 공을 쳐서 테이블에 있는 여섯 개의 구멍에 넣으면 되는 포켓볼 게임이다. 키보드의 왼쪽/오른쪽 방향키를 이용하여 흰 공을 치는 각도를 변경할 수 있고, 위쪽/아래쪽 방향키를 이용하여 흰 공을 치는 세기를 변경할 수 있다. 단, 검은색 공은 가장 마지막에 넣어야 한다.

만약 검은색 공을 마지막에 넣지 못하거나, 모든 공을 넣기 전에 흰색 공이 구멍에 들어간다면 게임에서 패배한다.

2. 코드 설명

```
# 당구공 객체 생성 함수
def initBall(p, r, mass, color):
    ball = sphere(pos=p, radius=r, color=color)
    ball.v = vec(0, 0, 0)
    ball.m = mass
    ball.f = vec(0, 0, 0)
    return ball

# 벽 객체 생성 함수
def initWall(pos, size):
    wall = box(pos=pos, size=size, color=color.green)
    return wall
```

당구공 객체와 벽 객체는 각각 함수를 정의하여 생성하였다.

당구공 객체를 생성하는 initBall()함수는 인자로 넘긴 위치, 반지름, 질량, 색상에 따라 sphere객체를 생성한 뒤, 반환한다.

벽 객체를 생성하는 initWall()함수는 인자로 넘긴 위치, 크기에 따라 판 모양의 box객체를 생성한 뒤, 반환한다.

```
# 키보드이벤트 함수
def on_keydown(event):
    global shoot
    global gamestate
    # 좌/우 방향키 : 각도 조절
    if (event.key == "left"):
        shoot.theta += 1
    elif (event.key == "right"):
        shoot.theta -= 1
    # 상/하 방향키 : 세기 조절
    elif (event.key == "up" and shoot.power < 0.8):
        shoot.power += 0.01
    elif (event.key == "down" and shoot.power > 0):
        shoot.power -= 0.01
    # Shift키 : 공 발사
    elif (event.key == "shift"):
        gamestate['ready'] = False
        gamestate['shoot'] = True
```

키보드 이벤트 함수이다. 전역 객체 shoot는 흰 공의 속도를 나타내는 화살표 객체이고, gamestate는 게임의 현재 상태를 나타내는 딕셔너리이다.

shoot객체에는 power속성과 theta속성이 있다. power속성은 흰 공을 치는 세기를 뜻하며, 화살표 객체의 길이가 power속성의 값에 비례하게 된다. theta속성은 흰 공을 치는 각도를 뜻한다.

gamestate의 key값은 'ready', 'shoot', 'gameover', 'gamelose', 'gamewin' 총 5개가 있고, 각 키값에 해당하는 value값은 각각 True, False, False, False, False로 초기화 되어있다. 'ready'가 True이면 현재 흰 공을 아직 치지 않은 상태. 즉, 방향키로 흰 공을 치는 각도와 세기를 조정하는 상태이고, 'shoot'이 True이면 흰 공을 친 상태. 즉, 공들이 운동하는 상태를 말한다. 게임이 종료되면 'gameover'는 True가 되고, 그와 동시에 'gamelose'나 'gamewin'중 하나의 값이 True로 바뀐다. 'gamelose'가 True이면 게임패배, 'gamewin'이 True이면 게임 승리를 뜻한다.

방향키의 왼쪽/오른쪽 버튼을 누르면 shoot객체의 theta속성의 값이 바뀌고, 위쪽/아래쪽 방향키를 누르면 shoot객체의 power속성의 값이 커지거나 작아진다. shift키를 누르면 게임의 현재 상태가 'ready'에서 'shoot'로 바뀐다.

```python
# 공끼리의 충돌 감지 함수
def collisionBall(b1, b2, e):
    r = b1.pos - b2.pos
    r_hat = norm(r)
    dist = mag(r)
    v_relm = dot(b1.v - b2.v, r_hat)
    tot_radius = b1.radius + b2.radius
    if v_relm > 0:
        return False
    if dist > tot_radius:
        return False
    if (dist <= tot_radius):
        j = -(1+e)*v_relm
        j = j/(1/b1.m+1/b2.m)
        b1.v = b1.v + j*r_hat/b1.m
        b2.v = b2.v - j*r_hat/b2.m
        b1.v *= 0.8
        b2.v *= 0.8

        return True
```

공끼리의 충돌을 감지하는 함수 collision_ball()이다. 인자로 두 공 객체와 반발계수e를 받아서, 두 공의 충돌을 구현한다.

두 공 사이의 거리가 두 공의 반지름을 더한 값보다 작다면 두 공은 충돌한 것이다. 벡터 r은 두 공의 위치벡터를 서로 뺀 값으로, 충돌면의 법선벡터가 된다. r_hat은 r의 단위벡터, dist는 벡터 r의 크기이다. v_relm은 두 공의 속도벡터의 차이와, r_hat을 내적한 값으로, 두 공의 상대속도를 나타낸다. 두 공의 상대속도가 0보다 클때는 서로 멀리 떨어져 운동하는 상황이므로 충돌을 고려하지 않아도 된다. 또한, 두 공 사이의 거리가 반지름의 합보다 클 때 역시 충돌을 고려하지 않아도 된다. 위의 두 조건을 만족하지 못했을 때, 충돌이 일어난다. 두 공의 충돌 후 속도는 충돌 전 속도와 충격량으로 표현할 수 있으므로, 각 물체의 충격량 계산식을 통해 충돌 후의 속도를 도출해낸다. 또한, 충돌 시 공 사이의 마찰력 때문에 속도가 조금 줄어들게 된다. 그래서 각 공의 속도 성분에 0.8을 곱해주었다.

```python
#공이 벽을 맞고 튕기게 하는 함수
def collisionWall(ball, table, e):
    if -table.size.x/2 > ball.pos.x - ball.radius:
        ball.pos.x = -table.size.x/2 + ball.radius
        ball.v.x = -e*ball.v.x

    if table.size.x/2 < ball.pos.x + ball.radius:
        ball.pos.x = table.size.x/2 - ball.radius
        ball.v.x = -e*ball.v.x

    if -table.size.z/2 > ball.pos.z - ball.radius:
        ball.pos.z = -table.size.z/2 + ball.radius
        ball.v.z = -e*ball.v.z

    if table.size.z/2 < ball.pos.z + ball.radius:
        ball.pos.z = table.size.z/2 - ball.radius
        ball.v.z = -e*ball.v.z
```

공이 벽을 맞고 튕기게 하는 함수 collisionWall()이다. 인자로 공 객체, table객체, 반발계수 e를 받는다.

공이 table의 왼쪽이나 오른쪽에 맞는 경우, 공의 x방향의 위치를 조금 조정해주고, 속도벡터의 x 성분의 부호를 바꿔주면 된다.
공이 table의 위쪽이나 아래쪽에 맞는 경우, 공의 z방향의 위치를 조금 조정해주고, 속도벡터의 z 성분의 부호를 바꿔주면 된다.

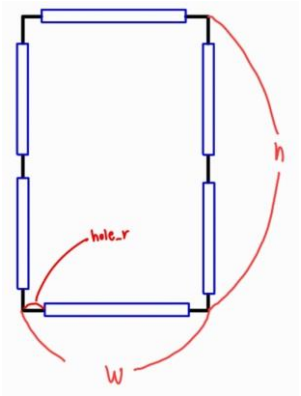
```
#공이 구멍으로 들어갔는지 여부를 체크하는 함수
def collisionHole(ball, hole_pos):
    global hole_r
    if(mag(ball.pos - hole_pos) < hole_r):
        return True
    else:
        return False
```

공이 구멍으로 들어갔는지 여부를 체크하는 함수 collisionHole()이다. 인자로 공 객체와 구멍 객체를 받아, 공과 구멍 사이의 거리보다 구멍의 반지름이 더 큰 경우 True를 리턴하고 아니면 False를 리턴한다.

```
#공의 위치를 업데이트 하는 함수
def updatePos(ball, dt):
    global g, uk
    ball.f = -uk*ball.m*mag(g)*norm(ball.v)
    ball.v += ball.f/ball.m*dt
    ball.pos += ball.v*dt
    if(mag(ball.v) < 0.01):
        ball.v = vec(0, 0, 0)
```

공 객체의 위치를 업데이트 하는 함수 updatePos()이다. 인자로 공 객체와 시간 간격 dt를 받아 해당 객체의 시간 간격에 따른 위치와 속도를 업데이트한다.

먼저, 공에 작용하는 알짜힘을 계산해야 한다. 현재 공에는 테이블과의 마찰력만이 존재하므로 알짜힘이 마찰력 그 자체가 된다. 공의 질량, 중력가속도, 마찰계수에 따라 공의 속도의 반대 방향으로 마찰력이 작용한다. 오일러-크로머 방식으로 공의 상태를 업데이트 해준다. 공의 속력이 0.01보다 작다면 공을 정지시킨다.



현재 테이블은 왼쪽 그림과 같은 상태이다.
가로길이 w, 세로길이 h이며, 테이블의 테두리에 벽이 총 6개 있다.
구멍의 반지름은 hole_r이다. 즉, 벽의 긴 변의 길이는 w-2*r이 된다.

w(=1.27m)와 h(=2.54m)는 실제 당구공 테이블 크기로 하였고,
hole_r(=0.07m)은 시뮬레이션 비율에 맞게 적당한 값으로 설정하였다.

미리 설정한 값과 함수를 이용하여 테이블과 벽 객체를 먼저 생성한다.

```
#공의 위치벡터 리스트
initpos_list = []

#공 객체 리스트
ball_list = []

#위치벡터 리스트 초기화
for i in range(5):
    for j in range(-i, i+1, 2):
        initpos_list.append(vec(r*j, 0, -sqrt(3)*i*r-0.3))

#흰 공(플레이어가 치는 공)을 공 객체 리스트에 가장 먼저 추가
ball_list.append(initBall(vec(0, 0, 0.8), r, m, color.white))

#공 객체 생성함수를 이용하여 공 객체 리스트에 공을 하나씩 추가함
for i in range(1, 16):
    ball_list.append(initBall(initpos_list[i-1], r, m, color_list[i-1]))
```

그 다음, 당구공 객체를 생성해준다. 당구공 객체 역시 실제 당구공과 비슷하게 반지름과 질량을 설정하였다. initpos_list는 공 15개의 초기 위치벡터가 저장된 리스트이고, color_list는 각 당구공 객체의 색상 값이 저장된 리스트이다. 이 리스트들을 이용하여 공 객체 리스트 ball_list에 객체를 하나씩 추가할 것이다.

가장 먼저 플레이어가 칠 흰색 공을 리스트에 추가한다. 그 다음, initpos_list와 color_list를 이용하여 ball_list에 총 15개의 공 객체를 추가해준다. 이러면 ball_list리스트의 0번째 인덱스의 객체는 항상 플레이어가 칠 공 객체를 나타낸다.

```
#흰 공의 속도를 나타내는 화살표객체
shoot = arrow(pos=ball_list[0].pos, color=color.white, shaftwidth=0.005)
shoot.power = 0.3
shoot.theta = -180
shoot.axis = vec(shoot.power*sin(radians(shoot.theta)),
                 0, shoot.power*cos(radians(shoot.theta)))
```

공의 속도벡터를 표시하는 shoot객체를 생성해준다. shoot객체의 power속성과 theta속성에 따라서, arrow객체를 생성하여 표시해준다.

```
while not gamestate['gameover']:
    rate(1/dt)

    #만약 공 리스트에 공 하나만 남아있다면(모든 공을 넣었다면)
    if (len(ball_list) == 1):
        #게임 승리, 종료
        gamestate['gameover'] = True
        gamestate['gamewin'] = True

    #게임이 'ready'상태라면
    if(gamestate['ready']):
        #카메라를 흰 공에 맞춘다.
        scene.center = ball_list[0].pos

        #흰 공을 치는 각도와 세기 업데이트
        shoot.pos = ball_list[0].pos
        shoot.visible = True
        shoot.axis = vec(shoot.power*sin(radians(shoot.theta)),
                         0, shoot.power*cos(radians(shoot.theta)))
        ball_list[0].v = 10*vec(shoot.power*sin(radians(shoot.theta)),
                                0, shoot.power*cos(radians(shoot.theta)))
```

이제 while루프를 돌려준다. while문은 gamestate의 gameover값이 False인 동안 진행된다.

만약 ball_list에 저장된 객체가 하나(흰 공)라면, gamestate의 gameover과 gamewin을 True로 바꿔준다.

만약 gamestate가 ready상태라면, 카메라를 흰 공에 맞춰주고, 방향키 입력에 따라 shoot객체와 흰 공의 속도 속성의 값을 업데이트 해준다. 이 상태에서 shift키를 누르면 shoot객체의 방향으로 흰 공을 치게 된다.

```
#게임이 'shoot'상태라면
elif(gamestate['shoot']):
    #화살표를 안보이게 설정
    shoot.visible = False

    #공 리스트 안의 각 객체의 위치를 업데이트 함
    for ball in ball_list:
        updatePos(ball, dt)

    #이중for문으로 공끼리의 충돌을 검사함.
    for b1 in ball_list:
        for b2 in ball_list:
            if(b1 == b2):
                continue
            collisionBall(b1, b2, 1)

    #공이 벽에 맞는지 검사
    for ball in ball_list:
        collisionWall(ball, table, 0.9)
```

gamestate가 shoot 상태일 경우 실행하는 코드이다. 먼저 shoot객체를 안보이게 설정해주고, ball_list안에 있는 공 객체들의 위치를 업데이트 해준다. 그 다음, 공과 다른 객체들 간의 충돌을 검사해준다.

먼저 공끼리의 충돌은 이중 for문으로 두 개의 공에 대하여 collisionBall함수를 호출하여 충돌을 검사해준다.

공과 벽의 충돌은 collisionWall()함수를 호출하여 공의 속도를 업데이트 해준다.

```
#공이 구멍에 들어갔는지 검사
for ball in ball_list:
    for hole in hole_list:
        if(collisionHole(ball, hole)):
            #만약 들어간 공이 흰공이라면
            if(ball == ball_list[0]):
                #게임 패배, 종료
                gamestate['gameover'] = True
                gamestate['gamelose'] = True
                ball.visible = False

            #만약 들어간 공이 검은 공이고 마지막에 넣은 것이 아니라면
            elif (ball.color == color.black and len(ball_list)>2):
                #게임 패배, 종료
                gamestate['gameover'] = True
                gamestate['gamelose'] = True
                ball.visible = False

            #그 외의 경우라면
            else:
                #공을 숨기고 리스트에서 삭제
                ball.visible = False
                ball_list.remove(ball)
```

공이 구멍에 들어갔는지 여부를 검사하는 부분은 이중 for문을 돌려 각 공 객체와 구멍 객체에 대하여 공 객체가 구멍에 들어갔는지 먼저 검사해준다. 공 객체가 구멍에 들어갔다면, 무슨 공이 들어갔는지 확인해 준다.

만약 들어간 공이 흰 공이라면, 게임 패배이므로 gamestate의 gameover와 gamelose값을 True로 변경해주고, 들어간 공이 안보이게 설정해준다.

만약 들어간 공이 검은색 공인데 마지막으로 넣은 것이 아니라면 흰 공을 넣었을 때와 같은 작업을 수행해준다.

이 외의 경우에는 들어간 공을 안보이게 설정해주고 ball_list에서 해당 공 객체를 삭제해준다.

```
#공이 모두 멈추었는지 확인하는 변수
stop = True
for ball in ball_list:
    if(mag(ball.v) > 0):
        stop = False

#공이 모두 멈추었다면
if stop:
    #게임의 현재 상태 업데이트
    gamestate['ready'] = True
    gamestate['shoot'] = False
```

stop변수는 모든 공이 멈추었는지 확인하는 변수이다. 각 공 객체에 대하여 하나라도 속도 속성이 0보다 큰 값이 있다면 stop은 False가 된다.

만약 모든 공이 멈추었다면, gamestate의 ready값을 True로 바꾸고, shoot값을 False로 바꿔준다.

```
#게임이 종료된 후
gameover_text = label(pos=ball_list[0].pos, height=30)

#게임을 패배했다면
if(gamestate['gamelose']):
    #"Game Over"텍스트를 띄운다.
    gameover_text.text = "Game Over!"

#게임을 승리했다면
elif(gamestate['gamewin']):
    #"You Win"텍스트를 띄운다.
    gameover_text.text = "You Win!"
```

while문이 종료된 후에, 게임을 승리했는지 패배했는지 구별하여 텍스트를 띄운다.

만약 게임을 패배했다면, 흰 공의 위치에 "Game Over!" 텍스트를 띄워준다.

만약 게임을 승리했다면, 흰 공의 위치에 "You Win!" 텍스트를 띄워준다.