Assignment1Extra: Collision Between Objects

20181200 권혁채
MAS2011-01
Prof. Yongduek Seo
2022-12-25

A. Introduction

The purpose of this assignment is to understand how to detect the collision between two objects, between an object and a wall, and determine how their movements change. We will be implementing the law of Conservation of Momentum and law of Conservation of Kinetic Energy. We will also assume that the space where the objects are moving is elastic (no loss of energy, no friction etc).

B. Remarks

Since we are assuming that our objects are in an elastic world, the total momentum of two colliding objects is preserved after the collision. Therefore, if we set m1, m2 as the mass of each object and v1, v2 as their velocity, in a one-dimensional space the following equation should be true. v1', v2' are the velocity of the objects after the collision.

$$m_1 v_1 + m_2 v_2 = m_1 {v_1}' + m_2 v_2'$$

Kinetic Energy should also be preserved, so the following equation should always be true.

$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 v_1'^2 + \frac{1}{2} m_2 v_2'^2$$

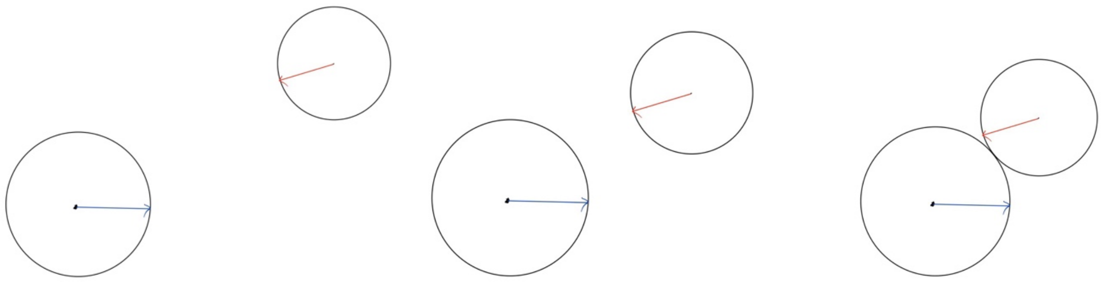
If we combine the two equations and derive equations with v1' and v2' on the right-hand side, we obtain the following equations for the velocity of m1 and m2 after the collision.

$$v_1' = \frac{v_1(m_1 - m_2) + 2m_2 v_2}{m_1 + m_2}$$

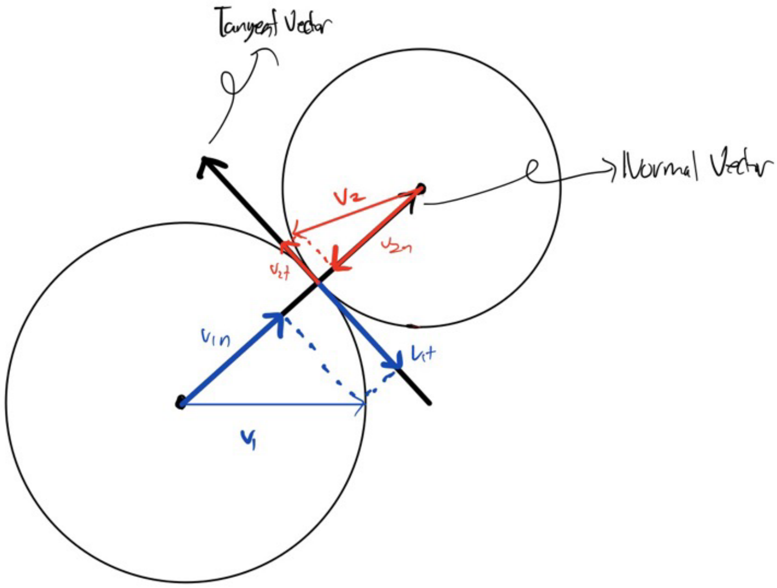
$$v_2' = \frac{v_2(m_2 - m_1) + 2m_1 v_1}{m_1 + m_2}$$

However, these are equations that only apply when the two objects are moving in a one-dimensional space, or in other words, the same direction. In order to determine velocities for objects colliding in two-dimensional space, we need to implement normal unit vectors and tangent unit vectors.

In order to better understand the process, let's visualize two objects of different mass colliding in a 2D space.



Now let's take a closer look at the moment of impact.



Let's assume that the big circle is C1, and the small circle is C2. Then each of their respective mass can be notated as m1, m2, velocity as vectors v1, v2, and center coordinates as x1, and x2. Since v1, and v2 are moving in completely unrelated directions, it would be convenient to convert them in relation to a unit normal vector and tangent vector. The unit normal vector is the vector of length 1 that passages through the center of both circles, and the unit tangent vector is a unit vector orthogonal to the normal vector. The unit normal vector can be found easily by calculating (x2-x1) / ||x2-x1||, and since the unit tangent vector is orthogonal to the normal vector, it is [-y component of unit normal vector, x component of unit normal vector].

Once we have obtained the unit normal and tangent vectors, we can project v1 and v2 to these vectors to get their counterparts v1n, v1t, v2n, v2t. Projection from one vector to another vector follows the equation p = a(aTb/aTa), but in this case a is a unit vector so all we have to do is find the dot product a•b to find its scalar velocity. Note that v1n + v1t = v1, v2n + v2t

= v2. Therefore, we can now calculate the change of velocity of each of these counterparts separately, and if we add the changed components together, we will get the vector for the changed velocity.

The tangential components v1t and v2t won't change after the collision since no force is applied in their direction, and since v1n and v2n are moving in the same direction, it is virtually colliding in a one-dimensional space. Therefore, we can find v1n' and v2n' using the 1D collision formulas from earlier.

$$v'_{1n} = \frac{v_{1n}(m_1 - m_2) + 2m_2 v_{2n}}{m_1 + m_2}$$

$$v'_{2n} = \frac{v_{2n}(m_2 - m_1) + 2m_1 v_{1n}}{m_1 + m_2}$$

Now that we have the changed scalar velocity of both normal vectors, we can convert them into vectors by multiplying them to their corresponding unit normal/tangent vectors. Hence:

$$\vec{v}'_{1n} = v'_{1n} * \overrightarrow{un} \qquad \vec{v}'_{1t} = v'_{1t} * \overrightarrow{ut} \qquad \vec{v}'_{2n} = v'_{2n} * \overrightarrow{un} \qquad \vec{v}'_{2t} = v'_{2t} * \overrightarrow{ut}$$

Finally, the new velocities will be the sum of each component.

$$\vec{v}'_1 = \vec{v}'_{1n} + \vec{v}'_{1t} \qquad \vec{v}'_2 = \vec{v}'_{2n} + \vec{v}'_{2t}$$

In the case when objects collide with a stationary wall in an elastic space, the incoming angle before the collision is the same as the resulting angle after the collision, so all we have to do is change the sign of only the x velocity if the object collides with walls on the right or left side, and change the sign of the y velocity if the object collides with walls on the top or bottom.

C. Programming

1. Creating the Ball class

```
# You can also use random.randrange(200,500)
class Ball:
    def __init__(self,):
        self.x = np.random.randint(200,900)
        self.y = np.random.randint(200,500)
        self.radius = np.random.randint(45,85)
        self.mass = (self.radius)**2*pi
        self.dx = np.random.randint(8,18)
        self.dy = np.random.randint(8,18)
        self.color = (np.random.randint(0,256),np.random.randint(0,256),np.random.randint(0,256))

    def update(self,):
        self.x+=self.dx
        self.y+=self.dy

        if self.x + self.radius > WINDOW_WIDTH or self.x - self.radius < 0:
            self.dx *= -1
        if self.y + self.radius > WINDOW_HEIGHT or self.y - self.radius < 0:
            self.dy *= -1

    def draw(self, screen):
        pygame.draw.circle(screen, self.color, [self.x, self.y], self.radius, 0)
```

The Ball class will have an x, y position denoting the position of the center of each circle, dx, dy members denoting the velocity of the object, a color denoted in rgb form, and a mass which will be equivalent to the area of the circle. Since the x, y coordinates represent the location of the center of the circle, the update(self) function checks whether the object has collided with a wall by checking its x, y coordinates +/- its radius. Note that in an elastic space, the incoming angle before the collision is the same as the resulting angle after the collision, so when the object collides with the left or right wall the sign of dx is reversed, and when it hits the top or bottom wall the sign of dy is reversed. The position, radius, velocity, and color of the object will all be chosen at random.

```
def collide(ball, ball2):
    if ((ball.x - ball2.x)**2 + (ball.y - ball2.y)**2)**(1/2)<(ball.radius+ball2.radius):
        m1 = ball.mass
        m2 = ball2.mass
        v1 = np.array([ball.dx, ball.dy])
        v2 = np.array([ball2.dx, ball2.dy])
        x1 = np.array([ball.x, ball.y])
        x2 = np.array([ball2.x, ball2.y])
        dist = math.sqrt(np.dot(x2-x1, x2-x1))

        normUnit = (x2-x1)/dist
        tanUnit = np.array([(-1)*normUnit[1],normUnit[0]])
        v1Norm = np.dot(normUnit, v1)
        v1Tan = np.dot(tanUnit, v1)
        v2Norm = np.dot(normUnit, v2)
        v2Tan = np.dot(tanUnit, v2)

        v1Norm_Next = (v1Norm*(m1-m2) + 2*m2*v2Norm) / (m1 + m2)
        v2Norm_Next = (v2Norm*(m2-m1) + 2*m1*v1Norm) / (m1 + m2)

        v1Norm_Next_vec = v1Norm_Next*normUnit
        v2Norm_Next_vec = v2Norm_Next*normUnit
        v1Tan_Next_vec = v1Tan*tanUnit
        v2Tan_Next_vec = v2Tan*tanUnit


        v1Next = v1Norm_Next_vec + v1Tan_Next_vec
        v2Next = v2Norm_Next_vec + v2Tan_Next_vec

        ball.dx = v1Next[0]
        ball.dy = v1Next[1]
        ball2.dx = v2Next[0]
        ball2.dy = v2Next[1]
```

This is the collide funciton which takes two Ball objects as parameters, and calculates their next velocity based on their current velocity and mass. It is the same process mentioned above, but written in code form. In order to carry out various vector calculations effectively, we have imported numpy as np. Note that the velocity of each ball only changes when the distance between the center of the two balls is smaller than the sum of their radius. This is how we detect when two circular objects have collided.

```python
listOfBalls = []
for i in range(2):
    ball = Ball()
    if i == 1:
        while(((listOfBalls[0].x - ball.x)**2 + (listOfBalls[0].y - ball.y)**2)**(1/2)<(listOfBalls[0].radius+ball.radius)):
            ball.x = np.random.randint(100,800)
            ball.y = np.random.randint(100,600)
    listOfBalls.append(ball)
```

Next, when we create the two Ball objects, we will make sure that they don't share any space when they are spawned, since this may cause additional problems.

```python
# 게임 반복 구간
while not done:
    # 이벤트 반복 구간
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    # 게임 로직 구간
    # 속도에 따라 원형 위치 변경 #state up date/logic update/parameter update

    for i in range(2):
        ball = listOfBalls[i]
        ball.update()

    collide(listOfBalls[0], listOfBalls[1])
    # 윈도우 화면 채우기
    screen.fill(WHITE)

    # 화면 그리기 구간
    # 공 그리기

    for i in range(2):
        ball = listOfBalls[i]
        ball.draw(screen)

    # 화면 업데이트
    pygame.display.flip()

    # 초당 60 프레임으로 업데이트
    clock.tick(60) # 60 frames per sec
                   # ball dx = 4
                   # ball velocity = 4 pixels per frame, so 240 frames per second

# 게임 종료
pygame.quit()
```

The rest of the code is almost identical to that of Assignment1, except for the fact that there are no Player-Controllable Objects, and that the function collide() is called after the update of each ball.

D. Improvements

Although collide() does serve its purpose, we can't help but notice that it has an awful lot of steps. Therefore, it would be more efficient if we could find a way to simplify the calculation process. According to 'Reducible', an educational YouTube channel on computer science, the calculations for velocities after collision can be simplified as follows.

$$\hat{v}_1 = v_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle v_1 - v_2, C_1 - C_2 \rangle}{||C_1 - C_2||^2}(C_1 - C_2)$$

$$\hat{v}_2 = v_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle v_2 - v_1, C_2 - C_1 \rangle}{||C_2 - C_1||^2}(C_2 - C_1)$$

"Building Collision Simulations: An Introduction to Computer Graphics" *YouTube*, uploaded by Reducible, 20 Jan 2021, https://www.youtube.com/watch?v=eED4bSkYCB8

Therefore the collide() function can be simplified as below.

```
def collide2(ball, ball2):
    if ((ball.x - ball2.x)**2 + (ball.y - ball2.y)**2)**(1/2)<(ball.radius+ball2.radius):
        m1 = ball.mass
        m2 = ball2.mass
        v1 = np.array([ball.dx, ball.dy])
        v2 = np.array([ball2.dx, ball2.dy])
        x1 = np.array([ball.x, ball.y])
        x2 = np.array([ball2.x, ball2.y])
        dist1 = np.dot(x1-x2, x1-x2)
        dist2 = np.dot(x2-x1, x2-x1)

        v1Next = v1 - (2*m2/(m1+m2))*np.dot(v1-v2, x1-x2)/dist1*(x1-x2)
        v2Next = v2 - (2*m1/(m1+m2))*np.dot(v2-v1, x2-x1)/dist2*(x2-x1)
        ball.dx = v1Next[0]
        ball.dy = v1Next[1]
        ball2.dx = v2Next[0]
        ball2.dy = v2Next[1]
```

This function is much more efficient as it has significantly less steps.

References

"Building Collision Simulations: An Introduction to Computer Graphics" *YouTube*, uploaded by Reducible, 20 Jan. 2021, https://www.youtube.com/watch?v=eED4bSkYCB8

Berchek, Chad. "2-Dimensional Elastic Collisions without Trigonometry", *Vobarian Software*, 3 Aug. 2009, https://www.vobarian.com/collisions/2dcollisions2.pdf