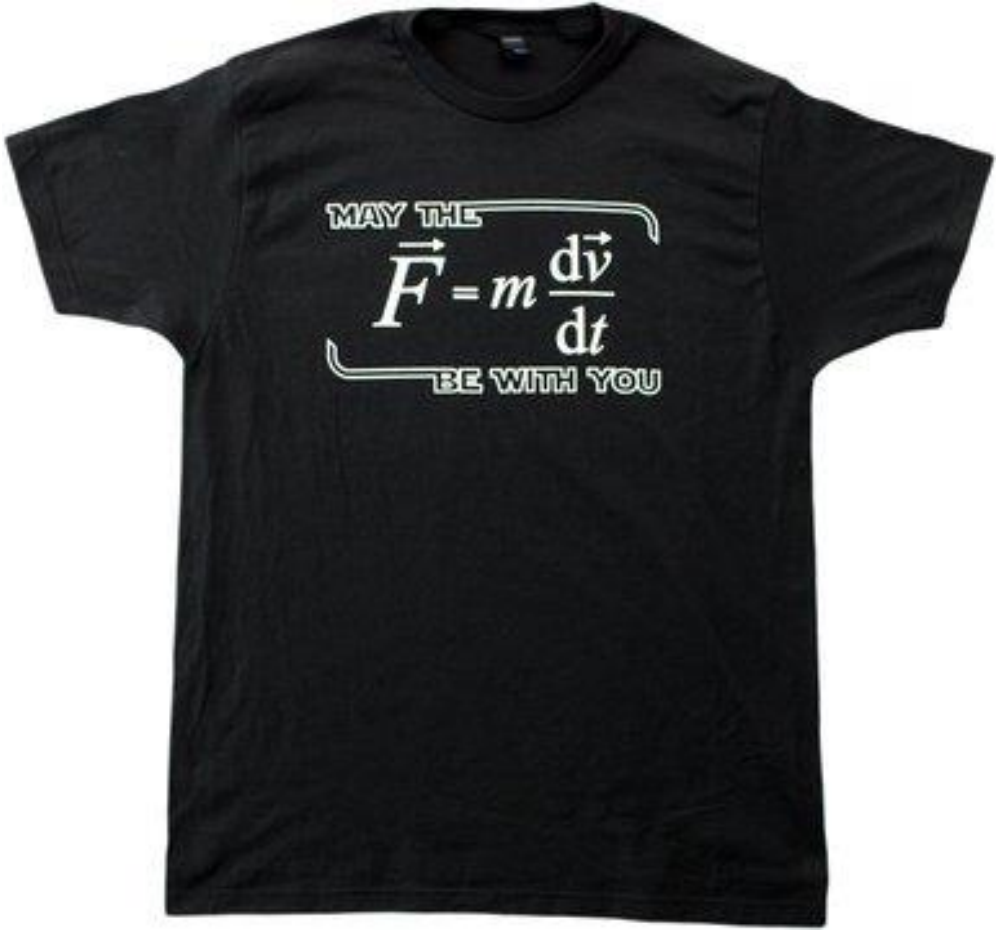
Linear Momentum, Impulse and Collision

#6

Serdar ARITAN

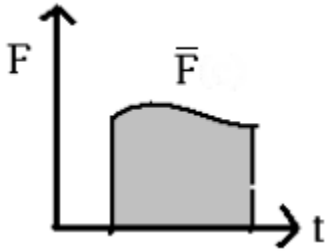
Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

PHYSICS in COMPUTER ANIMATIONS and GAMES



Linear Momentum

The linear momentum, $\vec{p}$, of an object is simply the mass of the object, m, multiplied by its velocity, $\vec{v}$.



$$\vec{p} = m\vec{v}$$

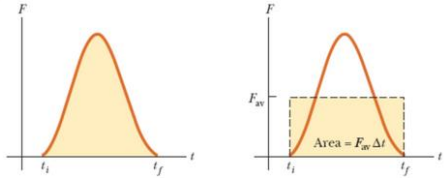
$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = \frac{d\vec{p}}{dt}$$

$$\int_{t_0}^{t_1} \vec{F}\,dt = \vec{p}_1 - \vec{p}_0$$

$$\bar{F} = \int \vec{F}\,dt$$

linear impulse of force

In the equation, $\vec{p}_0$ is the initial value of linear momentum and $\vec{p}_1$ is the momentum at the end of the time interval being considered. Equation indicates that a change in momentum of an object is equal to the integral of the net external force on the object as a function of time. The integration of the force with respect to time is known as **the linear impulse of force**.



$$\bar{F} = \vec{p}_1 - \vec{p}_0 = m(\vec{v}_1 - \vec{v}_0)$$

The force due to collision is known as an **impulsive force**. The magnitude of the impulsive force is usually so much larger than any other forces (gravity, drag, etc.) acting on the object during the collision, that all other forces can be ignored during the collision.

An important feature of impulsive force and linear impulse of force is that they act normal to the point of impact. As we shall see in a little while, the change in velocity due to a collision occurs normal to the point of impact as well. Newton's third law applies to linear impulses. If one object exerts a linear impulse on another, the second object will exert an equal and opposite impulse on the first object.

PHYSICS in COMPUTER ANIMATIONS and GAMES

An analysis of the collision between two objects depends on the momentum of the objects, but collisions can also be analyzed in terms of energy. Energy and momentum are always conserved in a collision, no matter what happens. Momentum is easy to deal with because there is only "one form" of momentum, but you do have to remember that momentum is a vector. Energy is tricky because it has many forms, the most troublesome being heat, but also sound and light. If kinetic energy is conserved in a collision, it is called an elastic collision. In an elastic collision, the total kinetic energy is conserved because the objects in question "bounce perfectly" like an ideal elastic. An inelastic collision is one where some of the of the total kinetic energy is transformed into other forms of energy, such as sound and heat. Generally speaking, the harder the objects are that collide, the closer the collision will be to being elastic.

PHYSICS in COMPUTER ANIMATIONS and GAMES

The collision of two marbles, for instance, will be a nearly elastic collision. On the other hand, the collision of a beanbag on the floor will be an inelastic collision. The coefficient, *e*, is known as the **coefficient of restitution** and has a value between 0 and 1. If *e* = 1, the pre- and post-collision relative velocities are equal, meaning that the collision is elastic. On the other hand, if *e* = 0, the post-collision relative velocity is zero (meaning that the objects are stuck together) and the collision is completely inelastic.

PHYSICS in COMPUTER ANIMATIONS and GAMES

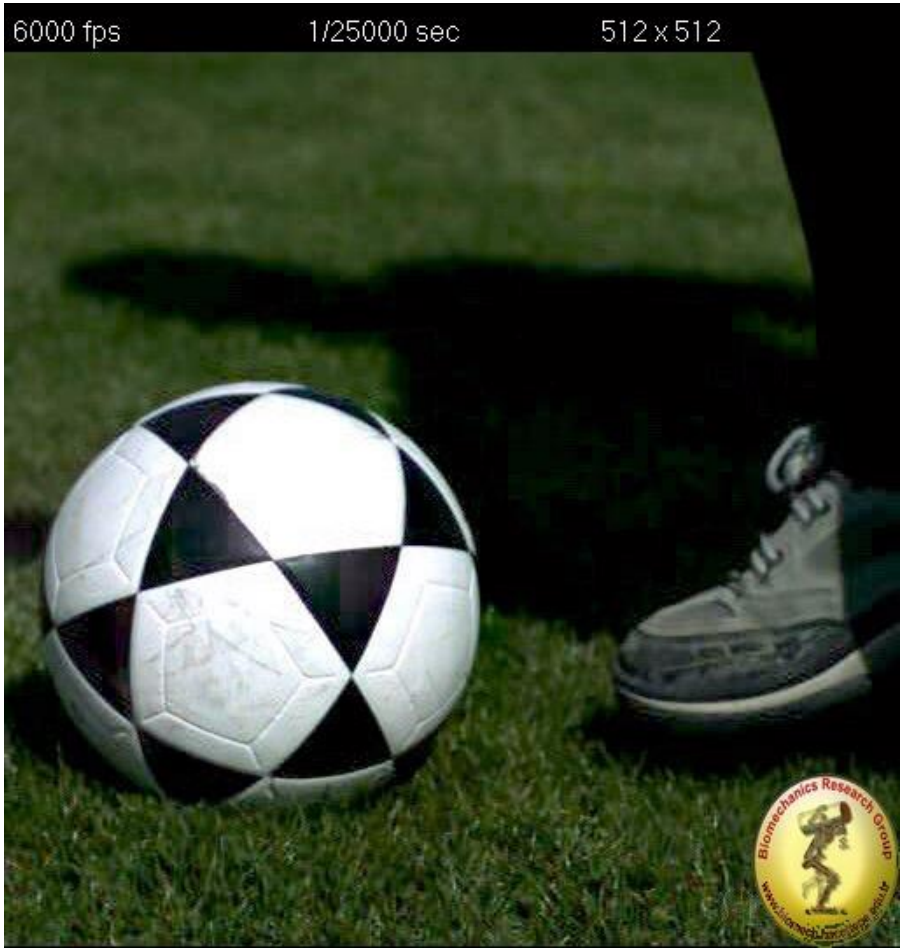
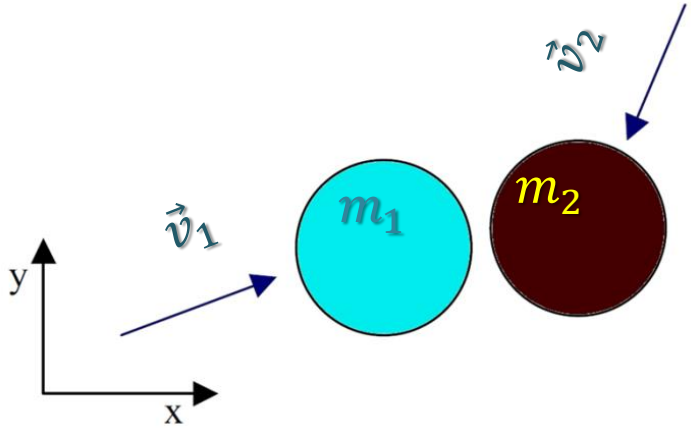
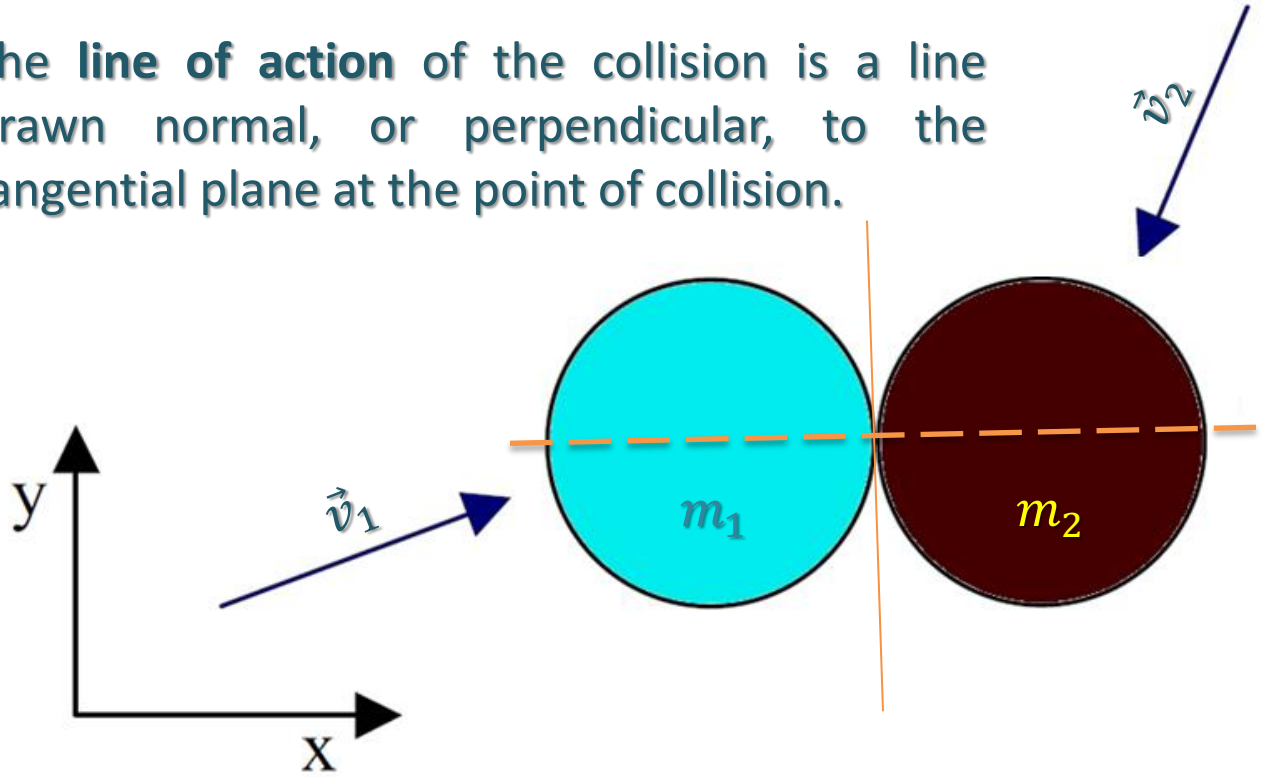


Figure below shows a general two-body collision in the x-y plane. The objects have some initial velocities $\vec{v}_1$ and $\vec{v}_2$ and masses m_1 and m_2.



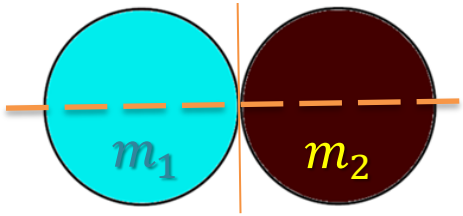
When they collide, the two objects will experience an impulse of force due to the collision. The magnitude of the impulse will be equal for both objects but will act in opposing directions. The geometric line along which the impulse acts is called the **line of action** for the collision.

The **line of action** of the collision is a line drawn normal, or perpendicular, to the tangential plane at the point of collision.



The linear impulse of force caused by the collision changes the velocity of the objects. Since the line of action for the collision is parallel to the x-axis, the linear impulse in the y direction is equal to zero.

$$\bar{F}_x = m(\vec{v}'_{1x} - \vec{v}_{1x})$$
$$0 = m(\vec{v}'_{1y} - \vec{v}_{1y})$$



In Equation, the pre-collision velocity components of object 1 are $\vec{v}_{1x}$ and $\vec{v}_{1y}$, and the post-collision velocity components are $\vec{v}'_{1x}$ and $\vec{v}'_{1y}$. The collision produces an equal-but opposite linear impulse applied to object 2.

$$-\bar{F}_x = m(\vec{v}'_{2x} - \vec{v}_{2x})$$
$$0 = m(\vec{v}'_{2y} - \vec{v}_{2y})$$

$$m(\vec{v}'_{1x} - \vec{v}_{1x}) + m(\vec{v}'_{1x} - \vec{v}_{1x}) = 0$$
$$\vec{v}'_{1y} = \vec{v}_{1y}$$
$$\vec{v}'_{2y} = \vec{v}_{2y}$$

three equations

If you look at Equation, you will see that momentum is conserved in both the x- and y-directions, but only the velocities in the x-direction change as a result of the collision. Looking more closely at the velocity expressions in Equation, there is still a problem. There are four unknowns, $\vec{v}'_{1x}, \vec{v}'_{1y}, \vec{v}'_{2x}, \vec{v}'_{2y}$ but only three equations. In order to solve for the post-collision velocities, an additional equation is introduced that relates the relative pre- and post-collision velocities of the two spheres along the line of action of the collision.

$$e(\vec{v}_{1x} - \vec{v}_{2x}) = -(\vec{v}'_{1x} - \vec{v}'_{2x})$$

The coefficient, **e**, is known as **the coefficient of restitution** and has a value between 0 and 1.

Combining Equations, expressions can be obtained for the post-collision velocities along the line of action.

$$\vec{v}'_{1x} = \frac{m_1 - em_2}{m_1 + m_2}\vec{v}_{1x} + \frac{(1+e)m_2}{m_1 + m_2}\vec{v}_{2x}$$

$$\vec{v}'_{2x} = \frac{(1+e)m_1}{m_1 + m_2}\vec{v}_{1x} + \frac{m_2 - em_1}{m_1 + m_2}\vec{v}_{2x}$$

We can see from Equation that the post-collision velocities along the line of action of the collision are a function of the pre-collision velocities along the line of action, the masses of the two objects, and the coefficient of restitution. The velocities in the y-direction, perpendicular to the line of action of the collision, are unaffected by the collision.

Equations in matrix form.

$$\begin{bmatrix} \vec{v}'_{1x} \\ \vec{v}'_{2x} \end{bmatrix} = \begin{bmatrix} \frac{m_1 - em_2}{m_1 + m_2} & \frac{(1+e)m_2}{m_1 + m_2} \\ \frac{(1+e)m_1}{m_1 + m_2} & \frac{m_2 - em_1}{m_1 + m_2} \end{bmatrix} \begin{bmatrix} \vec{v}_{1x} \\ \vec{v}_{2x} \end{bmatrix}$$

$$\begin{bmatrix} \vec{v}'_{1x} \\ \vec{v}'_{2x} \end{bmatrix} = \left(\frac{1}{m_1 + m_2} \right) \begin{bmatrix} m_1 - em_2 & (1+e)m_2 \\ (1+e)m_1 & m_2 - em_1 \end{bmatrix} \begin{bmatrix} \vec{v}_{1x} \\ \vec{v}_{2x} \end{bmatrix}$$

A General Two-Dimensional Collision

This problem, like all linear collision problems, can be broken down into four steps.

1. Determine the line-of-action vector for the collision.
2. Use a rotation matrix to determine the velocity components along the line of action and normal to it.
3. Compute the post-collision velocities.
4. Rotate the post-collision velocities back to the original Cartesian coordinate system.

A General Two-Dimensional Collision

Use a rotation matrix to determine the velocity components along the line of action and normal to it.

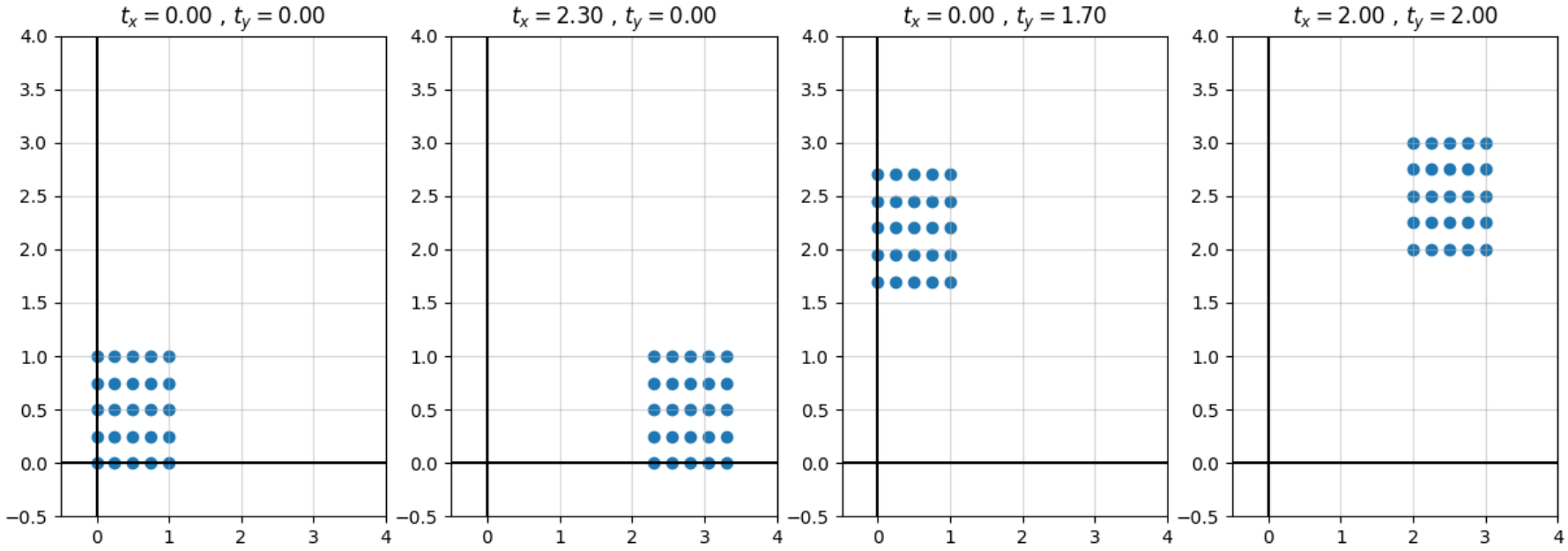
Translation

$$x' = x + t_x$$
$$y' = y + t_y$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

PHYSICS in COMPUTER ANIMATIONS and GAMES

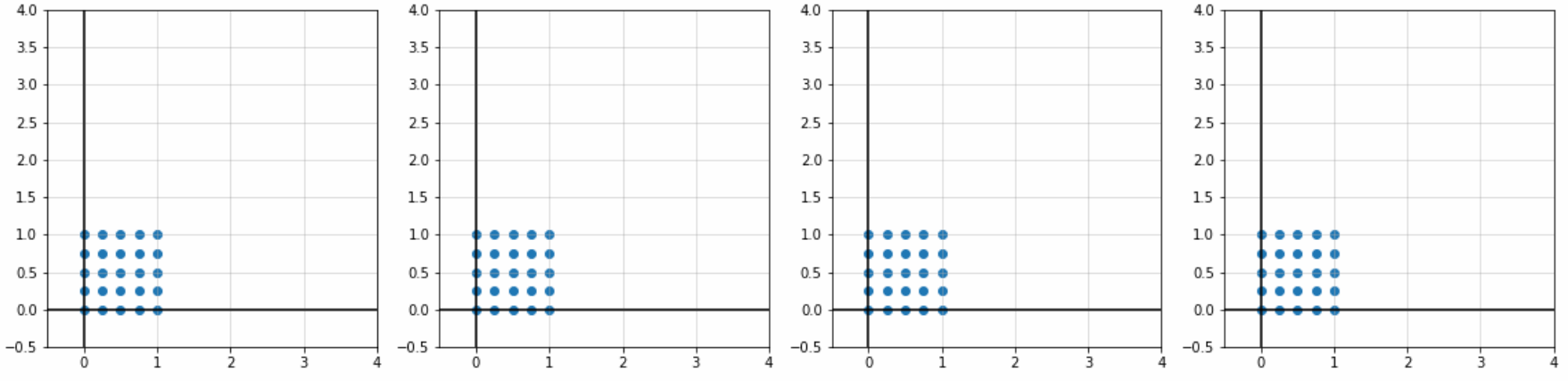
```
import matplotlib.pyplot as plt
import numpy as np

X, Y = np.mgrid[0:1:5j, 0:1:5j]
x, y = X.ravel(), Y.ravel()

def trans_translate(x, y, tx, ty):
    T = [[1, 0, tx],
         [0, 1, ty],
         [0, 0, 1 ]]
    T = np.array(T)
    P = np.array([x, y, [1]*x.size])
    return np.dot(T, P)

fig, ax = plt.subplots(1, 4)
T_ = [[0, 0], [2.3, 0], [0, 1.7], [2, 2]]

for i in range(4):
    tx, ty = T_[i]
    x_, y_, _ = trans_translate(x, y, tx, ty)
    ax[i].scatter(x_, y_)
    ax[i].set_title(r'$t_x={0:.2f}$ , $t_y={1:.2f}$'.format(tx, ty))
    ax[i].set_xlim([-0.5, 4])
    ax[i].set_ylim([-0.5, 4])
    ax[i].grid(alpha=0.5)
    ax[i].axhline(y=0, color='k')
    ax[i].axvline(x=0, color='k')

plt.show()
```


$t_x = 0.00$, $t_y = 0.00$
$t_x = 2.30$, $t_y = 0.00$
$t_x = 0.00$, $t_y = 1.70$
$t_x = 2.00$, $t_y = 2.00$

Translation

$$x' = x + t_x$$
$$y' = y + t_y$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Scaling

Relative to the point (p_x, p_y)

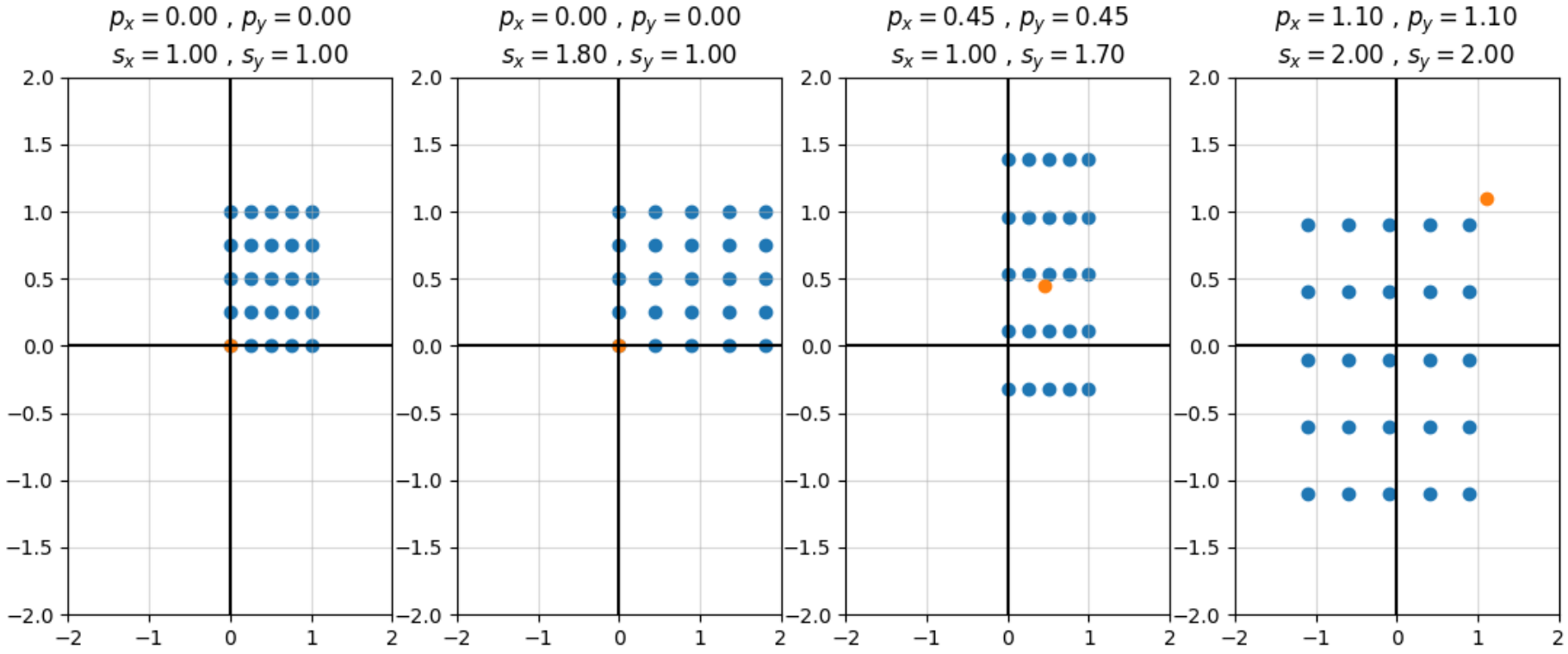
$$x' = s_x(x - p_x) + p_x = s_x x + p_x(1 - s_x)$$
$$y' = s_y(y - p_y) + p_y = s_y y + p_y(1 - s_y)$$

using homogeneous matrix

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & p_x(1 - s_x) \\ 0 & s_y & p_y(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

```
def trans_scale(x, y, px, py, sx, sy):
    T = [[sx, 0 , px*(1 - sx)],
         [0 , sy, py*(1 - sy)],
         [0 , 0 , 1          ]]
    T = np.array(T)
    P = np.array([x, y, [1]*x.size])
    return np.dot(T, P)

fig, ax = plt.subplots(1, 4)

S_ = [[1, 1], [1.8, 1], [1, 1.7], [2, 2]]
P_ = [[0, 0], [0, 0], [0.45, 0.45], [1.1, 1.1]]

for i in range(4):
    sx, sy = S_[i]; px, py = P_[i]
    x_, y_, _ = trans_scale(x, y, px, py, sx, sy)
    ax[i].scatter(x_, y_)
    ax[i].scatter(px, py)
    ax[i].set_title(r'$p_x={0:.2f}$ , $p_y={1:.2f}$'.format(px, py) + '\n'
                    r'$s_x={0:.2f}$ , $s_y={1:.2f}$'.format(sx, sy))

    ax[i].set_xlim([-2, 2])
    ax[i].set_ylim([-2, 2])
    ax[i].grid(alpha=0.5)
    ax[i].axhline(y=0, color='k')
    ax[i].axvline(x=0, color='k')

plt.show()
```

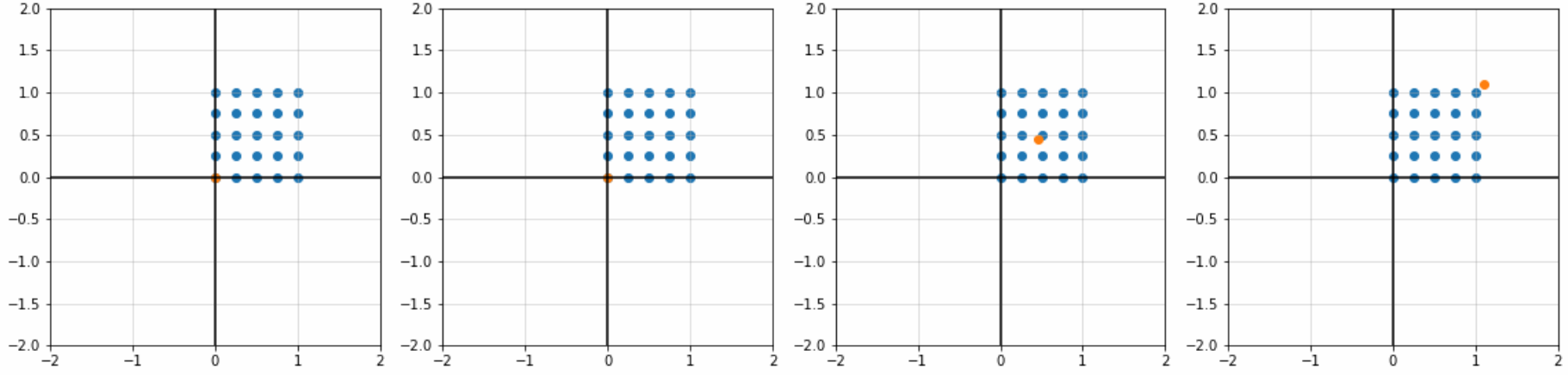
Scaling


$p_x = 0.00$, $p_y = 0.00$
$s_x = 1.00$, $s_y = 1.00$
$p_x = 0.00$, $p_y = 0.00$
$s_x = 1.80$, $s_y = 1.00$
$p_x = 0.45$, $p_y = 0.45$
$s_x = 1.00$, $s_y = 1.70$
$p_x = 1.10$, $p_y = 1.10$
$s_x = 2.00$, $s_y = 2.00$

Scaling

$$x' = s_x(x - p_x) + p_x = s_x x + p_x(1 - s_x)$$
$$y' = s_y(y - p_y) + p_y = s_y y + p_y(1 - s_y)$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & p_x(1 - s_x) \\ 0 & s_y & p_y(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Rotation

Relative to the point (p_x, p_y)

$$x' = (x - p_x)\cos\beta - (y - p_y)\sin\beta + p_x = x\cos\beta - y\sin\beta + p_x(1 - \cos\beta) + p_y\sin\beta$$
$$y' = (x - p_x)\sin\beta + (y - p_y)\cos\beta + p_y = x\sin\beta + y\cos\beta + p_y(1 - \cos\beta) - p_x\sin\beta$$

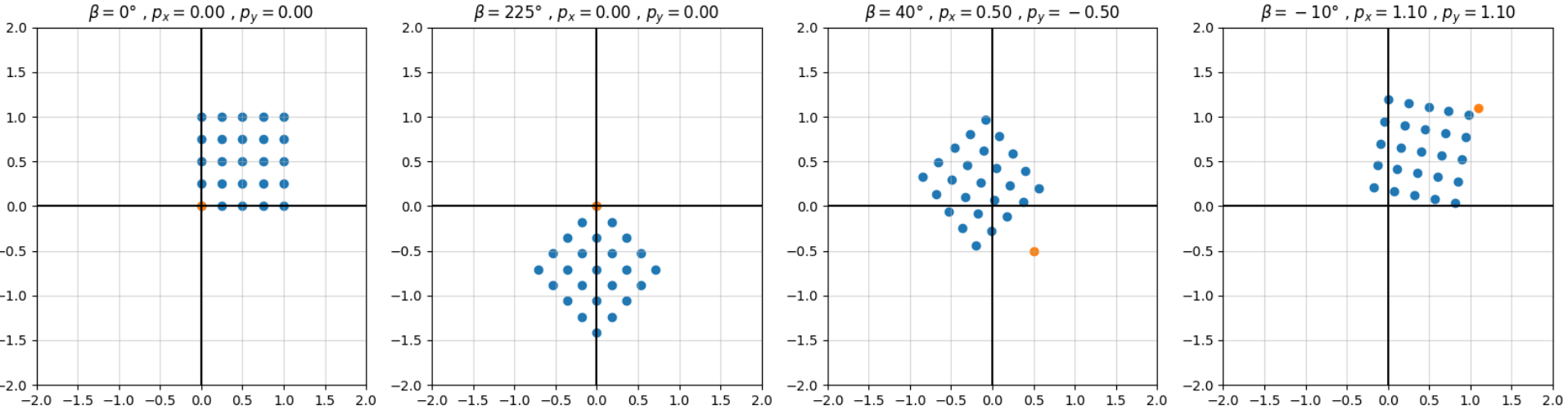
using homogeneous matrix

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & p_x(1-\cos\beta) + p_y\sin\beta \\ \sin\beta & \cos\beta & p_y(1-\cos\beta) - p_x\sin\beta \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
def trans_rotate(x, y, px, py, beta):
    beta = np.deg2rad(beta)
    T = [[np.cos(beta), -np.sin(beta), px*(1 - np.cos(beta)) + py*np.sin(beta)],
         [np.sin(beta),  np.cos(beta), py*(1 - np.cos(beta)) - px*np.sin(beta)],
         [0           ,  0           , 1                                      ]]
    T = np.array(T)
    P = np.array([x, y, [1]*x.size])
    return np.dot(T, P)

fig, ax = plt.subplots(1, 4)

R_ = [0, 225, 40, -10]
P_ = [[0, 0], [0, 0], [0.5, -0.5], [1.1, 1.1]]

for i in range(4):
    beta = R_[i]; px, py = P_[i]
    x_, y_, _ = trans_rotate(x, y, px, py, beta)
    ax[i].scatter(x_, y_)
    ax[i].scatter(px, py)
    ax[i].set_title(r'$\beta={0}°$ , $p_x={1:.2f}$ , $p_y={2:.2f}$'.format(beta, px, py))

    ax[i].set_xlim([-2, 2])
    ax[i].set_ylim([-2, 2])
    ax[i].grid(alpha=0.5)
    ax[i].axhline(y=0, color='k')
    ax[i].axvline(x=0, color='k')

plt.show()
```

Rotation


$\beta = 0°$, $p_x = 0.00$, $p_y = 0.00$
$\beta = 225°$, $p_x = 0.00$, $p_y = 0.00$
$\beta = 40°$, $p_x = 0.50$, $p_y = -0.50$
$\beta = -10°$, $p_x = 1.10$, $p_y = 1.10$

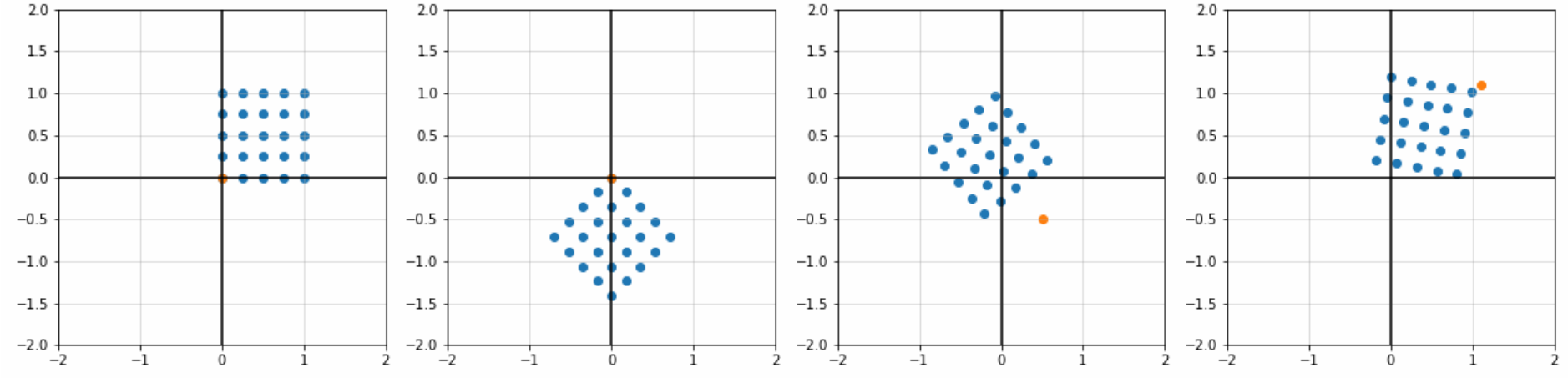
Rotation

Relative to the point (p_x, p_y)

$$x' = (x - p_x)\cos\beta - (y - p_y)\sin\beta + p_x = x\cos\beta - y\sin\beta + p_x(1 - \cos\beta) + p_y\sin\beta$$
$$y' = (x - p_x)\sin\beta + (y - p_y)\cos\beta + p_y = x\sin\beta + y\cos\beta + p_y(1 - \cos\beta) - p_x\sin\beta$$

using homogeneous matrix

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & p_x(1-\cos\beta) + p_y\sin\beta \\ \sin\beta & \cos\beta & p_y(1-\cos\beta) - p_x\sin\beta \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Shearing

Relative to the point (p_x, p_y)

$$x' = x + \lambda_x(y - p_x) = x + \lambda_x y - \lambda_x p_x$$
$$y' = y + \lambda_y(x - p_y) = y + \lambda_y x - \lambda_y p_y$$

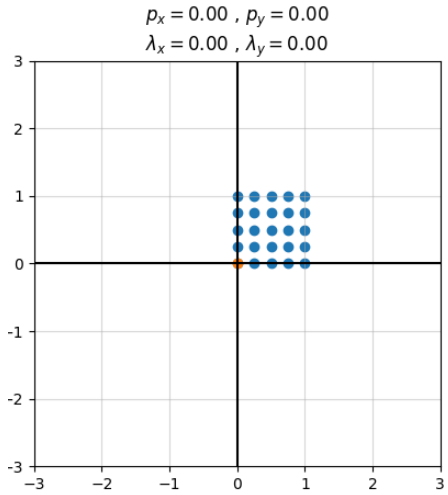
using homogeneous matrix

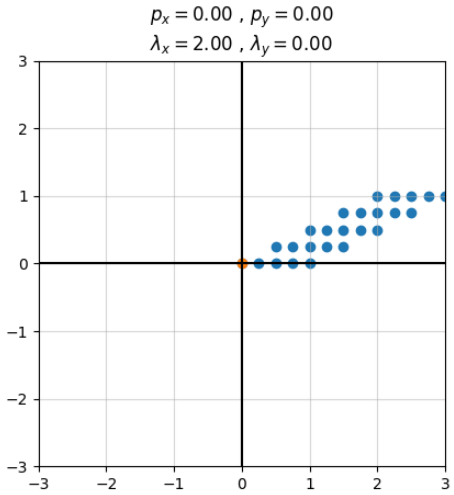
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \lambda_x & -\lambda_x p_x \\ \lambda_y & 1 & -\lambda_y p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

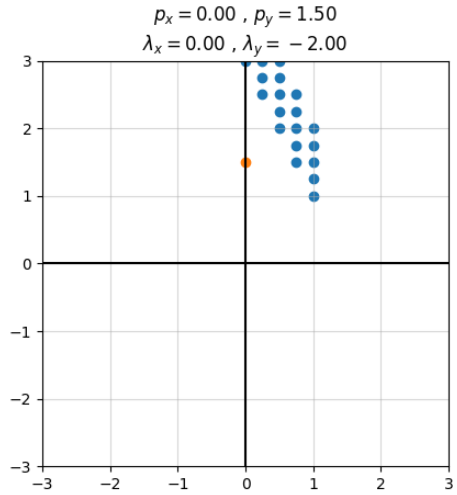
PHYSICS in COMPUTER ANIMATIONS and GAMES

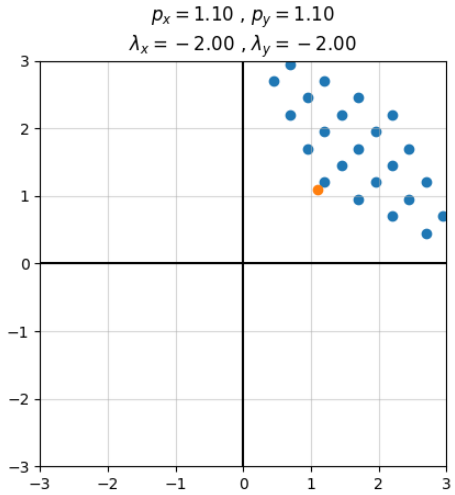
```
def trans_shear(x, y, px, py, lambdax, lambday):
    T = [[1       , lambdax, -lambdax*px],
         [lambday, 1       , -lambday*py],
         [0       , 0       ,  1          ]]
    T = np.array(T)
    P = np.array([x, y, [1]*x.size])
    return np.dot(T, P)

fig, ax = plt.subplots(1, 4)

L_ = [[0, 0], [2, 0], [0, -2], [-2, -2]]
P_ = [[0, 0], [0, 0], [0, 1.5], [1.1, 1.1]]

for i in range(4):
    lambdax, lambday = L_[i]; px, py = P_[i]
    x_, y_, _ = trans_shear(x, y, px, py, lambdax, lambday)
    ax[i].scatter(x_, y_)
    ax[i].scatter(px, py)
    ax[i].set_title(r'$p_x={0:.2f}$ , $p_y={1:.2f}$'.format(px, py) + '\n'
                    r'$\lambda_x={0:.2f}$ , $\lambda_y={1:.2f}$'.format(lambdax, lambday))

    ax[i].set_xlim([-3, 3])
    ax[i].set_ylim([-3, 3])
    ax[i].grid(alpha=0.5)
    ax[i].axhline(y=0, color='k')
    ax[i].axvline(x=0, color='k')

plt.show()
```

Shearing

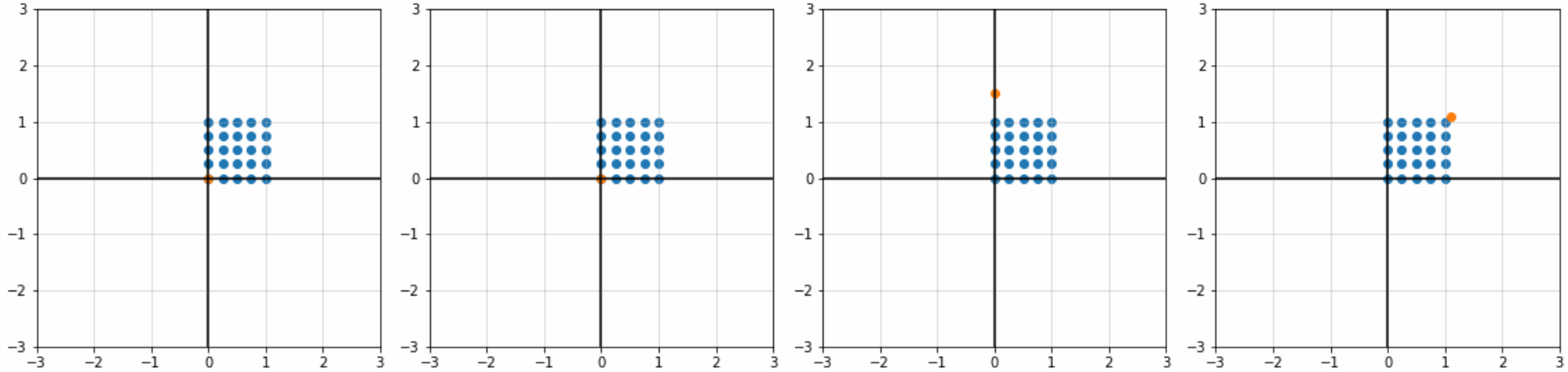

$p_x = 0.00$, $p_y = 0.00$
$\lambda_x = 0.00$, $\lambda_y = 0.00$


$p_x = 0.00$, $p_y = 0.00$
$\lambda_x = 2.00$, $\lambda_y = 0.00$

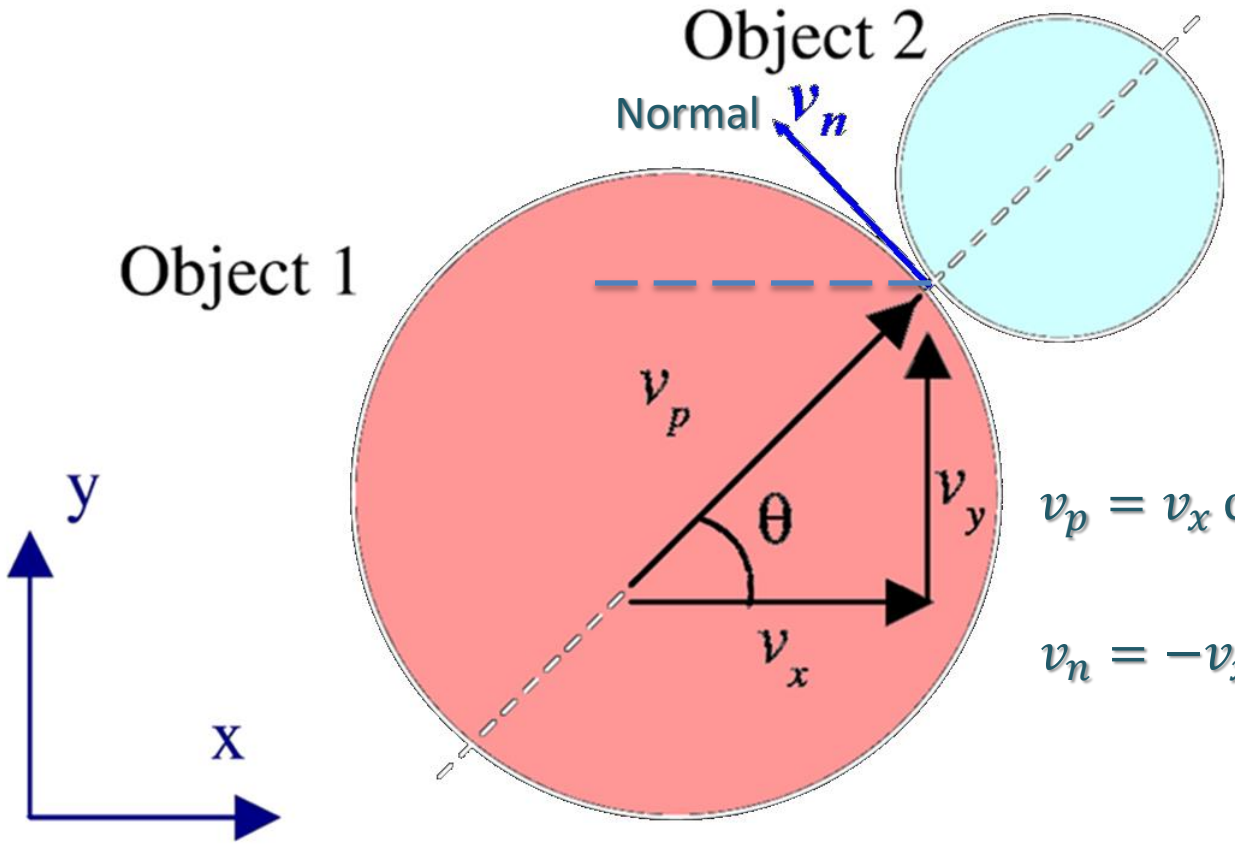

$p_x = 0.00$, $p_y = 1.50$
$\lambda_x = 0.00$, $\lambda_y = -2.00$


$p_x = 1.10$, $p_y = 1.10$
$\lambda_x = -2.00$, $\lambda_y = -2.00$

Shearing

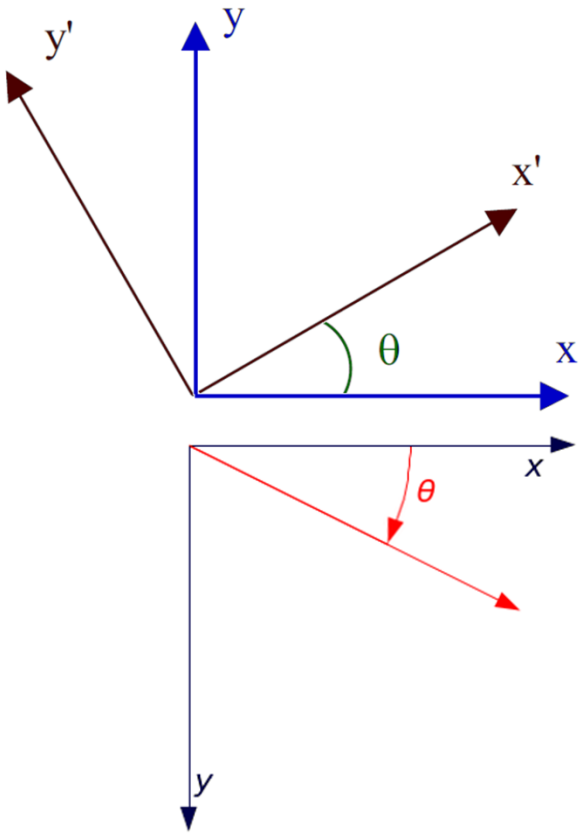


Determine the line-of-action vector for the collision



$$v_p = v_x \cos\theta + v_y \sin\theta$$

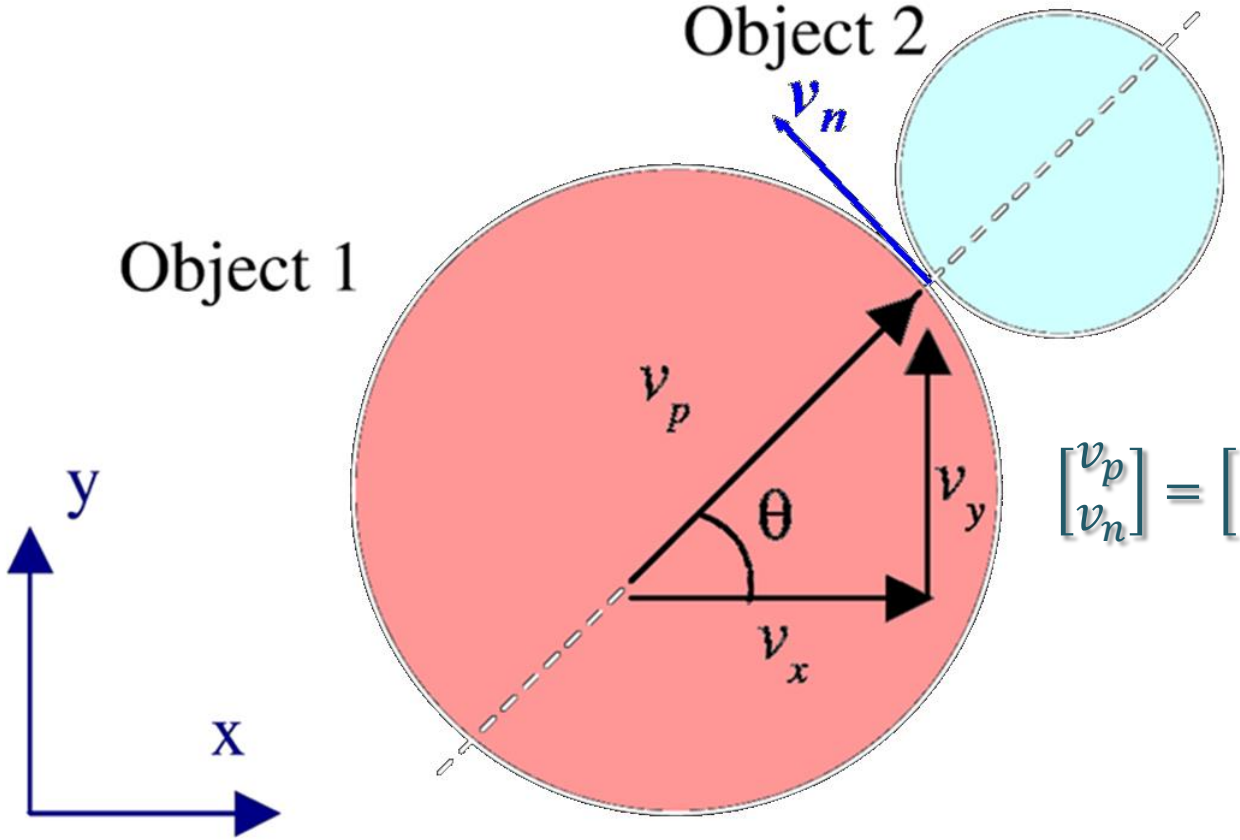
$$v_n = -v_x \sin\theta + v_y \cos\theta$$



$$R(\theta) = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

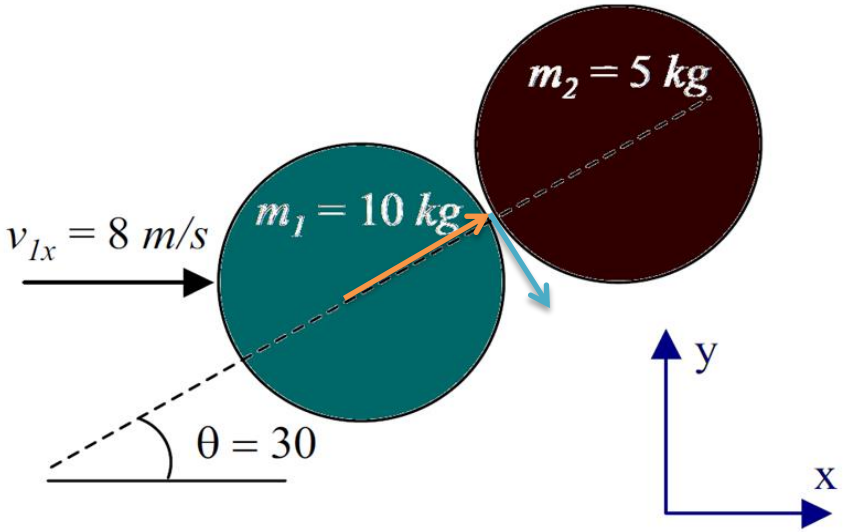
The direction of vector rotation is counterclockwise if θ is positive (e.g. 90°), and clockwise if θ is negative (e.g. -90°).

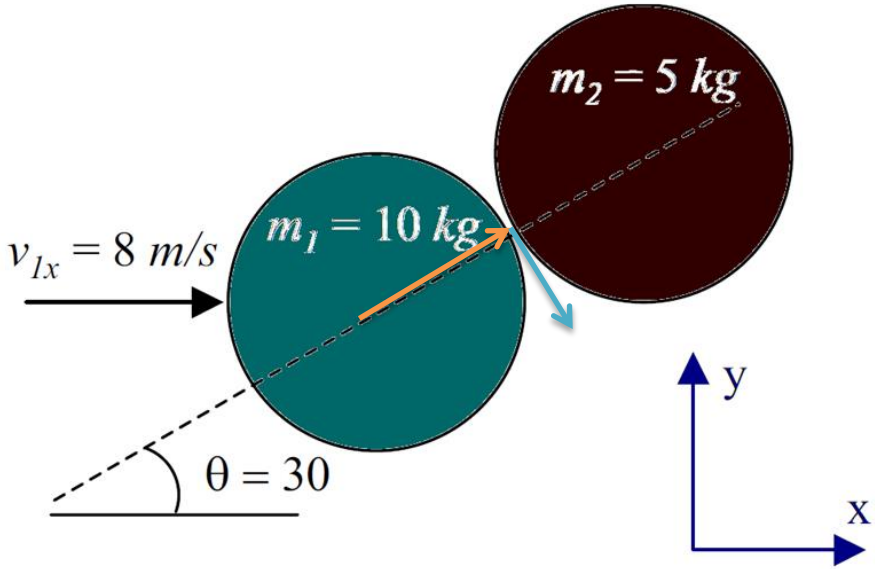
$$R(-\theta) = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



$$\begin{bmatrix} v_p \\ v_n \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

Two spheres collide. **Sphere 1** has a mass of **10 kg** and is traveling horizontally with a velocity v_{1x} = **8 m/s**. **Sphere 2** is stationary and has a mass of **5 kg**. The line of action for the collision is at an angle of **30 degrees** with respect to the x-axis. **The coefficient of restitution** between the two spheres is **0.9**. What will be the post-collision velocities of the two spheres?





$$\begin{bmatrix} v_p \\ v_n \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

$$\begin{bmatrix} v_{1p} \\ v_{1n} \end{bmatrix} = \begin{bmatrix} \cos 30 & \sin 30 \\ -\sin 30 & \cos 30 \end{bmatrix} \begin{bmatrix} v_{1x} \\ v_{1y} \end{bmatrix}$$

$$\begin{bmatrix} v_{1p} \\ v_{1n} \end{bmatrix} = \begin{bmatrix} \cos 30 & \sin 30 \\ -\sin 30 & \cos 30 \end{bmatrix} \begin{bmatrix} 8 \\ 0 \end{bmatrix}$$

The velocity components parallel and normal to the line of action can be computed. Sphere 2 is initially not moving, so its velocity components will be zero.

$$v_{1p} = 8\cos 30 + 0\sin 30 = 6.93\ ms^{-1}$$

$$v_{1n} = -8\sin 30 + 0\cos 30 = -4.00\ ms^{-1}$$

Compute the post-collision velocities for the two spheres

$$\begin{bmatrix} \vec{v}'_{1x} \\ \vec{v}'_{2x} \end{bmatrix} = \left(\frac{1}{m_1 + m_2} \right) \begin{bmatrix} m_1 - em_2 & (1+e)m_2 \\ (1+e)m_1 & m_2 - em_1 \end{bmatrix} \begin{bmatrix} \vec{v}_{1p} \\ \vec{v}_{2p} \end{bmatrix}$$

$$\begin{bmatrix} \vec{v}'_{1x} \\ \vec{v}'_{2x} \end{bmatrix} = \left(\frac{1}{10 + 5} \right) \begin{bmatrix} 10 - 0.9\text{x}5 & (1+0.9)5 \\ (1+0.9)10 & 5 - 0.9\text{x}10 \end{bmatrix} \begin{bmatrix} 6.93 \\ 0 \end{bmatrix}$$

$$\vec{v}'_{1x} = \frac{10 - 4.5}{15} 6.93 = 2.54\ ms^{-1}$$

$$\vec{v}'_{2x} = \frac{19}{15} 6.93 = 8.78\ ms^{-1}$$

The final step in the process is to rotate the post-collision velocities back to the standard Cartesian coordinate system.

$$\begin{bmatrix} v''_{1x} \\ v''_{1y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v'_{1x} \\ v'_{1n} \end{bmatrix} \qquad \begin{bmatrix} v''_{2x} \\ v''_{2y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v'_{2x} \\ v'_{2n} \end{bmatrix}$$

$$\begin{bmatrix} v''_{1x} \\ v''_{1y} \end{bmatrix} = \begin{bmatrix} \cos 30 & -\sin 30 \\ \sin 30 & \cos 30 \end{bmatrix} \begin{bmatrix} 2.54 \\ -4 \end{bmatrix} \qquad \begin{bmatrix} v''_{2x} \\ v''_{2y} \end{bmatrix} = \begin{bmatrix} \cos 30 & -\sin 30 \\ \sin 30 & \cos 30 \end{bmatrix} \begin{bmatrix} 8.78 \\ 0 \end{bmatrix}$$

$v''_{1x} = 2.45\cos 30 + 4\sin 30 = 4.1\ ms^{-1}$ $\qquad v''_{2x} = 8.78\cos 30 = 7.60\ ms^{-1}$

$v''_{1y} = 2.45\sin 30 - 4\cos 30 = -2.24\ ms^{-1}$ $\qquad v''_{2y} = 8.78\sin 30 = 4.39\ ms^{-1}$

After the collision, both spheres are moving in the positive x-direction, but sphere 1 has slowed down because part of its momentum was transferred to sphere 2 during the collision. Sphere 2 is traveling in the positive y-direction and sphere 1 is traveling in the negative y-direction.

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
import numpy as np

theta = np.radians(30)    # Line of action
m1 = 10                   # Sphere 1 has a mass of 10 kg
m2 = 5                    # Sphere 2 has a mass of 5 kg
e = 0.9                   # the coefficient of restitution
v1pre = np.matrix([[8], # Sphere 1 moving horizontally with a velocity 8 m/s.
                   [0]])
v2pre = np.matrix([[0], # Sphere 2 is stationary
                   [0]])

R1 = np.matrix([[np.cos(theta), np.sin(theta)],
                [-np.sin(theta), np.cos(theta)]])
R2 = np.matrix([[np.cos(theta), -np.sin(theta)],
                [np.sin(theta), np.cos(theta)]])

v1p, v1n = R1*v1pre
v2p, v2n = R1*v2pre
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Compute the post-collision velocities for the two spheres
v1pt, v2pt = np.multiply((1.0/(m1+m2)),\
np.matrix([[(m1-e*m2),m2*(1+e)],[m1*(1+e),(m2-e*m1)]])*np.vstack((v1p, v2p)))

# The final step in the process is to rotate the post-collision velocities
# back to the standard Cartesian coordinate system

v1post = R2*np.vstack((v1pt, v1n))
v2post = R2*np.vstack((v2pt, v2n))
```

Circle-Circle Collision Detection

Circle1 with center (x_1,y_1) and radius r_1;
Circle2 with center (x_2,y_2) and radius r_2.

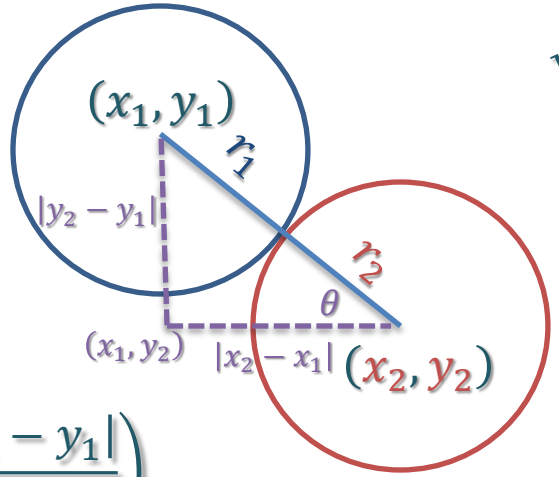
The most basic way of determining whether or not two circles have collided is

1. Find the distance between the centers of the two circles using the distance formula.
2. if the edges of the circles touch, the distance between the centers is r_1+r_2
   any greater distance and the circles don't touch or collide.
   any less and then do collide

So you can detect collision if $(x_2-x_1)^2 + (y_1-y_2)^2 <= (r_1+r_2)^2$

Calculate the point of collision

Circle1 with center (x_1,y_1) and radius r_1;
Circle2 with center (x_2,y_2) and radius r_2.



$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} = r_1 + r_2$$

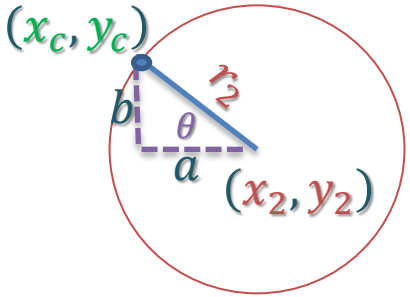
$$\sin\theta = \frac{|y_2 - y_1|}{r_1 + r_2}$$

$$\cos\theta = \frac{|x_2 - x_1|}{r_1 + r_2}$$

$$\tan\theta = \frac{|y_2 - y_1|}{|x_2 - x_1|}$$

$$\theta = \operatorname{atan}\left(\frac{|y_2 - y_1|}{|x_2 - x_1|}\right)$$

Calculate the point of collision (x_c, y_c)

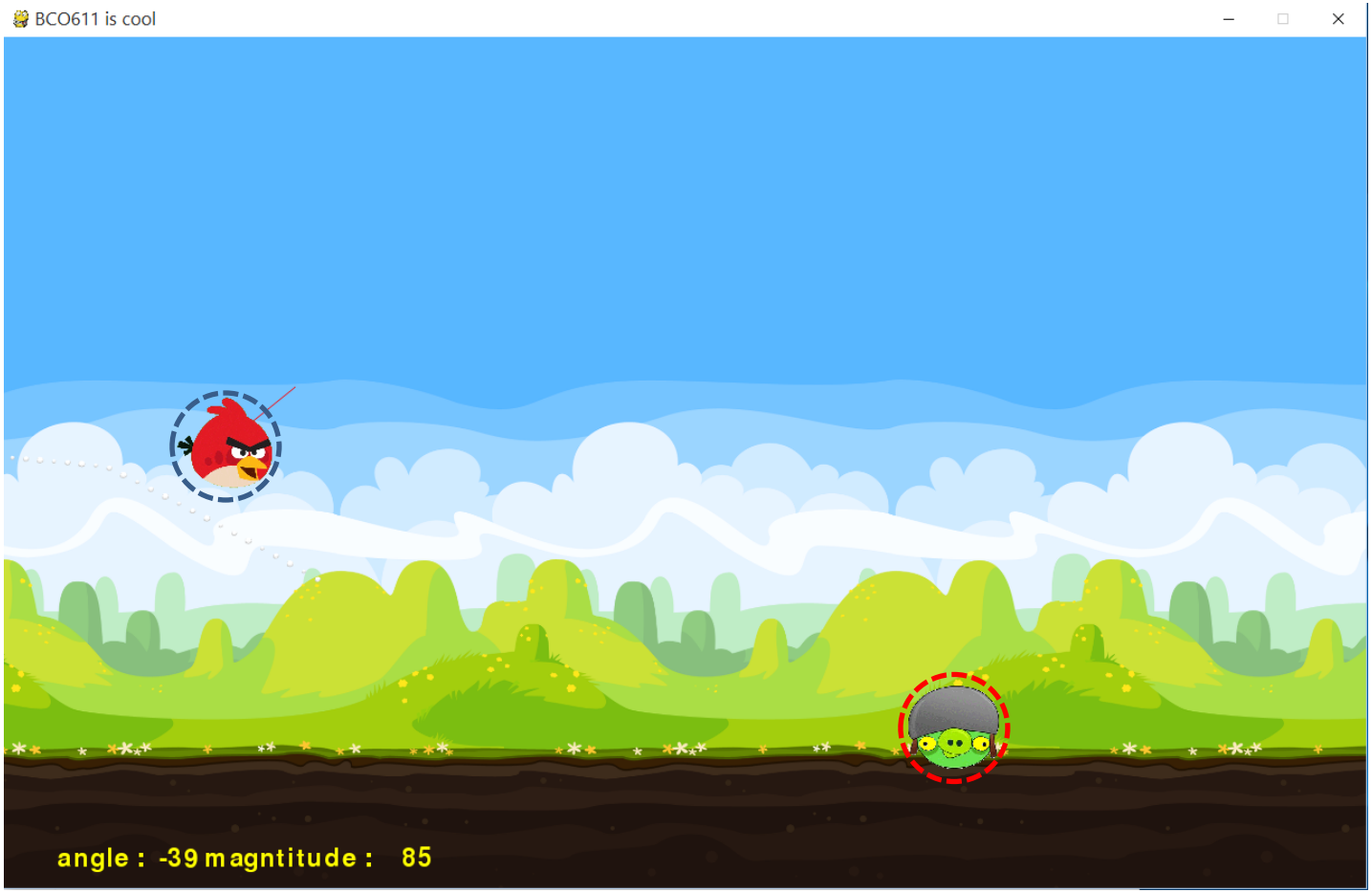


$$\cos\theta = \frac{a}{r_2} \qquad \sin\theta = \frac{b}{r_2}$$

$$a = r_2 \cos\theta \qquad b = r_2 \sin\theta$$

$$a = r_2 \frac{|x_2 - x_1|}{r_1 + r_2} \qquad b = r_2 \frac{|y_2 - y_1|}{r_1 + r_2}$$

$$(x_c, y_c) = (x_2 + a\,, y_2 + b)$$


BCO611 is cool
angle : -39 magntitude : 85

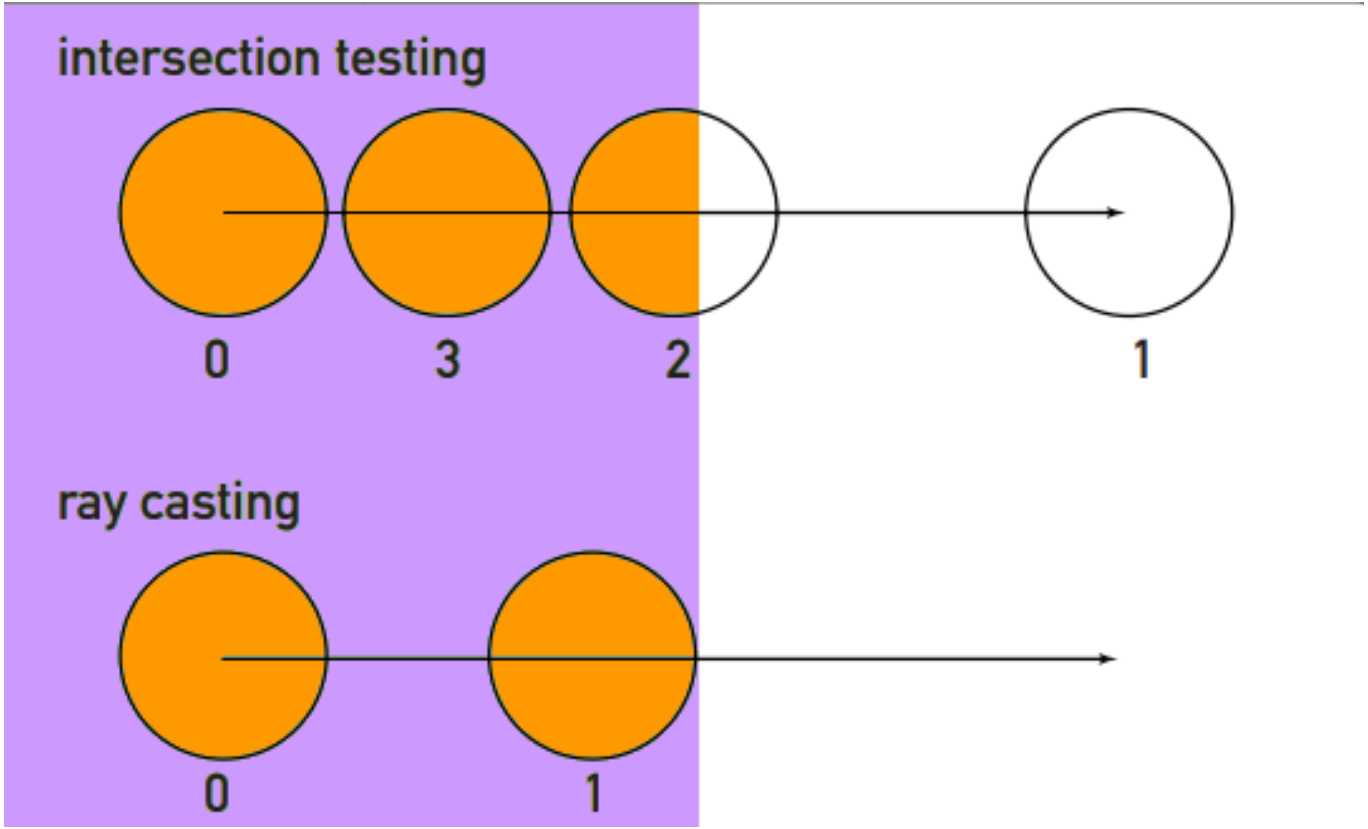
PHYSICS in COMPUTER ANIMATIONS and GAMES

$$\theta = \operatorname{atan}\left(\frac{|y_2 - y_1|}{|x_2 - x_1|}\right)$$

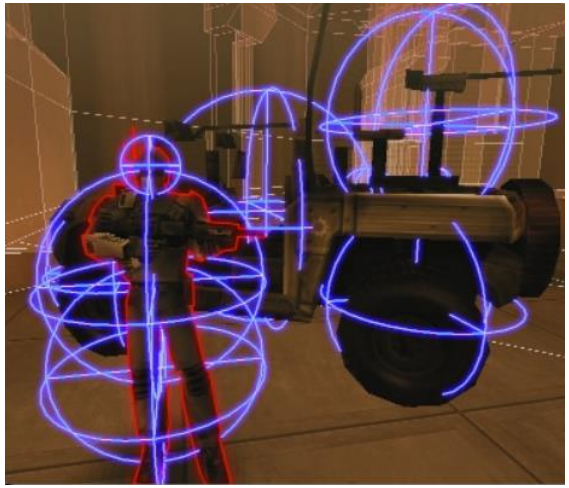
$$a = r_2 \frac{|x_2 - x_1|}{r_1 + r_2} \qquad b = r_2 \frac{|y_2 - y_1|}{r_1 + r_2}$$

$$(x_c, y_c) = (x_2 + a\,, y_2 + b)$$

Intersection Testing versus Ray Casting

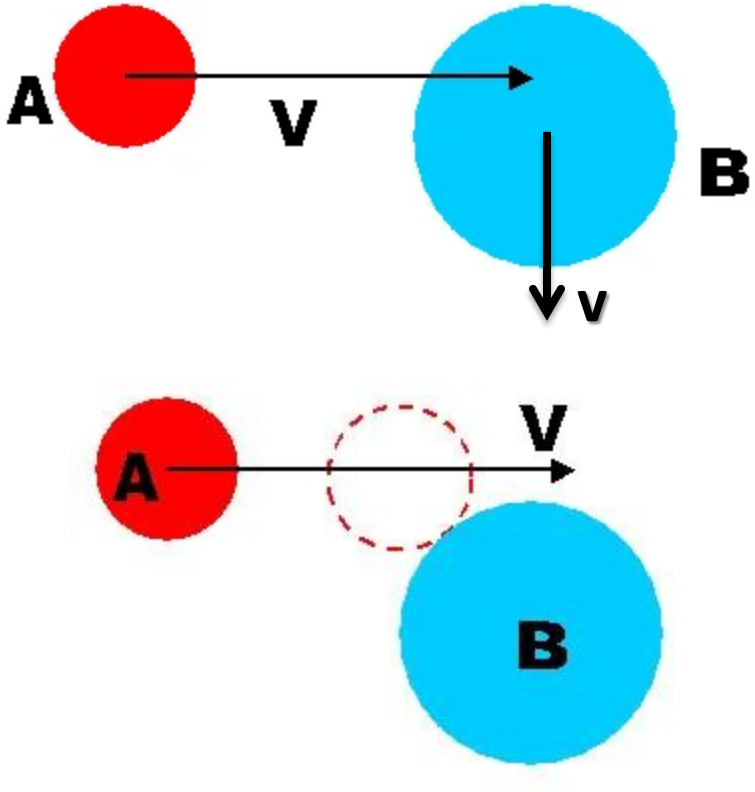


Tim Schroeders's article "**Collision Detection Using Ray Casting**", inthe August 2001 issue of Game Developer magazine focused on detecting collisions between spheres and polygons.



This is useful not only for games like pool where accurate collision of spheres is key, but also in games where characters and other mobile objects are bounded by spheres, these can be used to quickly determine if they have bumped into each other.

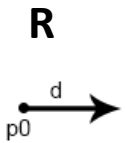
Collision Detection Using Ray Casting

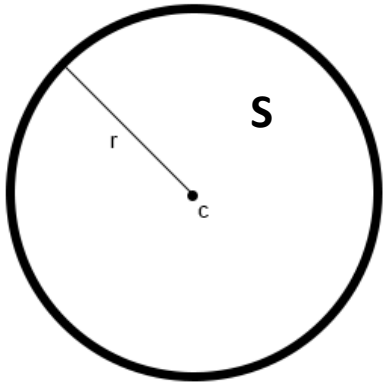


Collision Detection Using Ray Casting

Ray **R**, defined by:
- point **p0**
- normal **d**

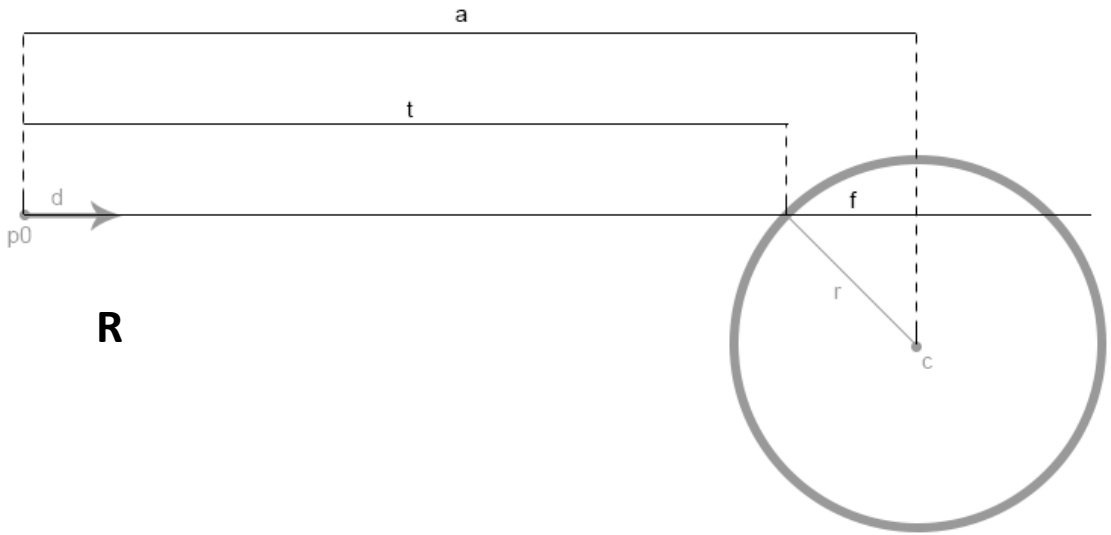
Sphere **S**, defined by:
- center point **c**
- radius **r**





Collision Detection Using Ray Casting

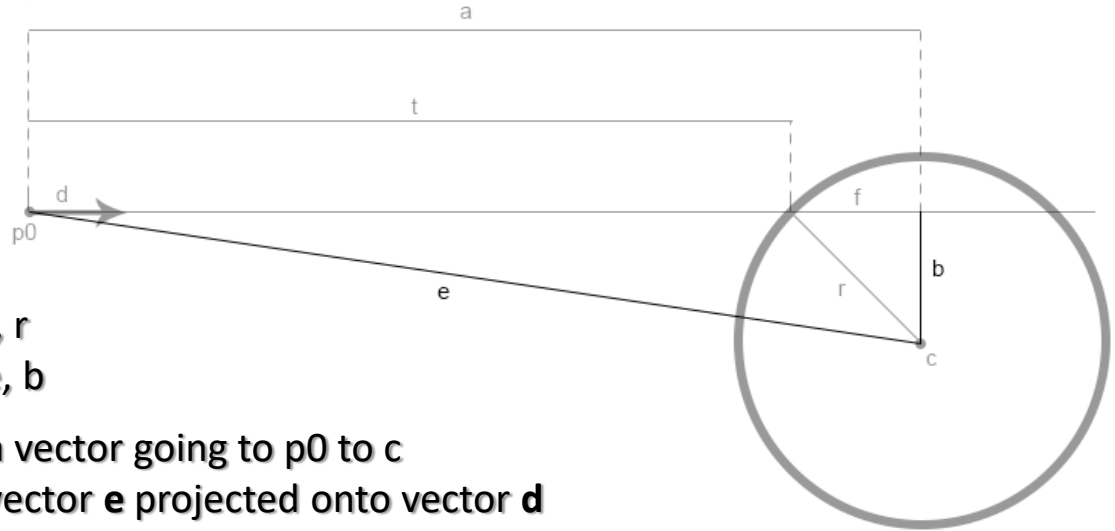
An intersection happens at time **t**, along ray **R**.



To find **t**, we need to first find **a** and **f**. The formula for **t** = a- f

Collision Detection Using Ray Casting

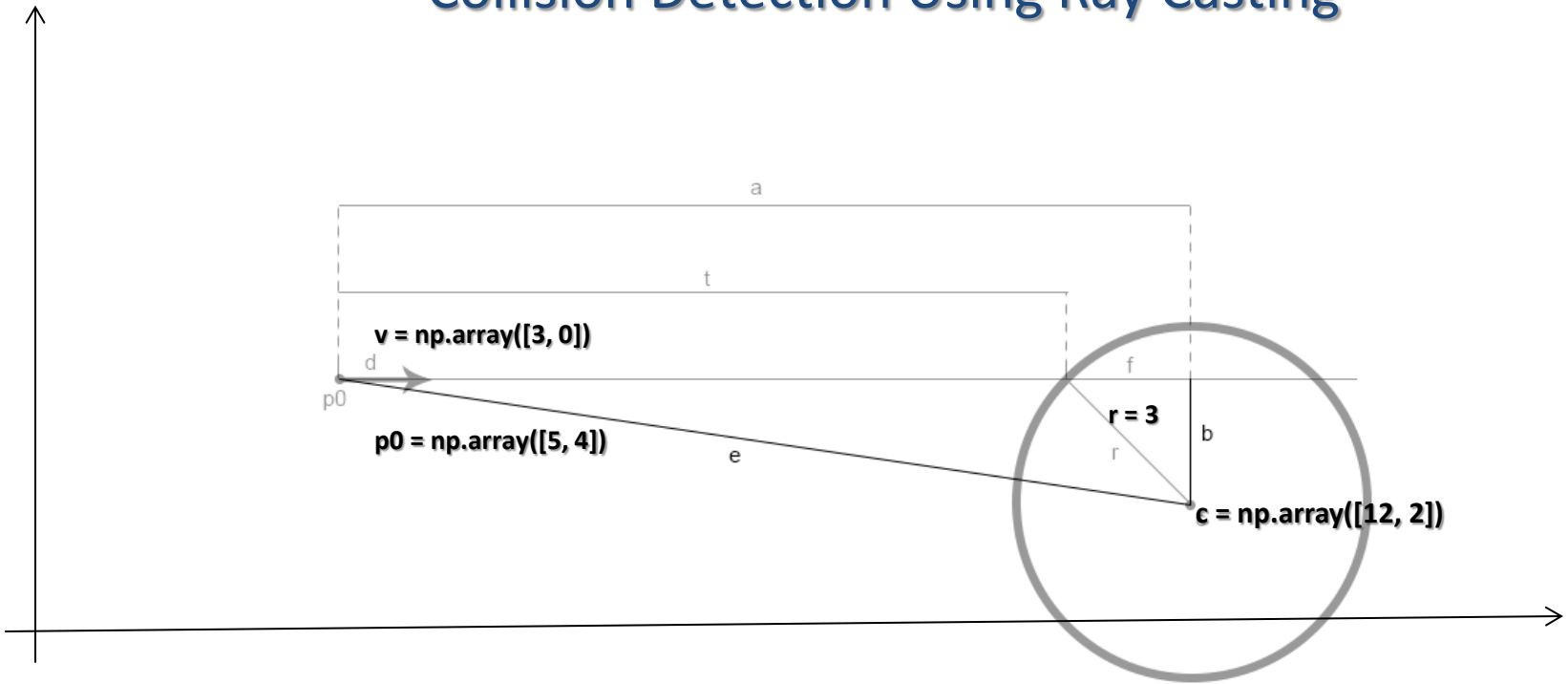
In order to find **a** and **f**, we need two more vectors, vector **e** and vector **b**. Vector **e** is the vector from **p0** to **c**. Vector **b** is one side of the right triangle formed by vectors **r**, **f** and **b**.



Triangle f, b, r
Triangle a, e, b

Vector **e** is a vector going to p0 to c
Vector **a** is vector **e** projected onto vector **d**
Vector **b** is given by the Pythagorean theorem

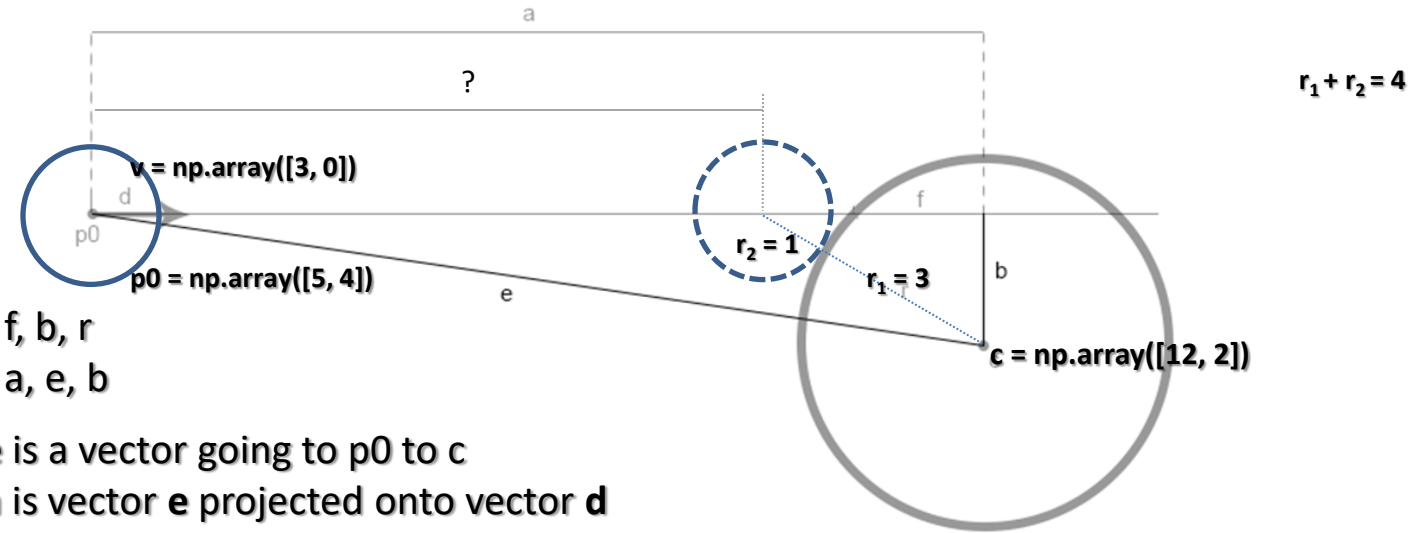
Collision Detection Using Ray Casting



Collision Detection Using Ray Casting

```
import numpy as np

p0 = np.array([5, 4])    # Ray Position;
v = np.array([3, 0])     # we will use velocity : d ray
normalized_v = v / np.sqrt(np.sum(v**2))
c = np.array([12, 2])    # Sphere Centre Position
r = 3                    # Sphere Radius

e = c - p0
esq = np.square(np.linalg.norm(e))
a = np.dot(e, normalized_v)
b = np.sqrt(esq - (a**2))
f = np.sqrt((r**2) - (b**2))

# No collision
if (r**2 - esq + a**2) < 0.0 :
    print(-1)     # -1 is invalid
    # Ray is inside
elif esq < r**2:
    print(a + f) # Just reverse direction
else: # Normal intersection
    print( a - f)
```

```
4.76393202250021
```

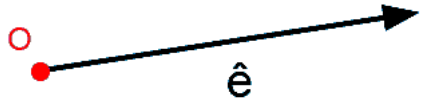


Triangle f, b, r
Triangle a, e, b

Vector **e** is a vector going to p0 to c
Vector **a** is vector **e** projected onto vector **d**
Vector **b** is given by the Pythagorean theorem

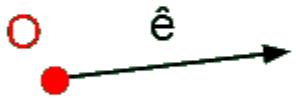
Ray-Object Intersections: Primary rays

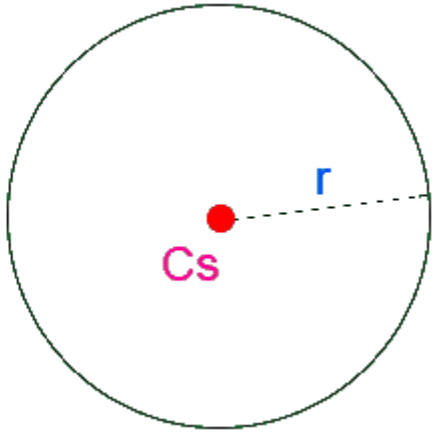
```
import matplotlib.pyplot as plt
import numpy as np

## Ray
O = np.array([0, 0])          # Origin point
e_ = np.array([0.5, 0.5])     # Ray direction
e_ /= np.linalg.norm(e_)      # Unit vector of e_
```



Ray-Object Intersections: Primary rays

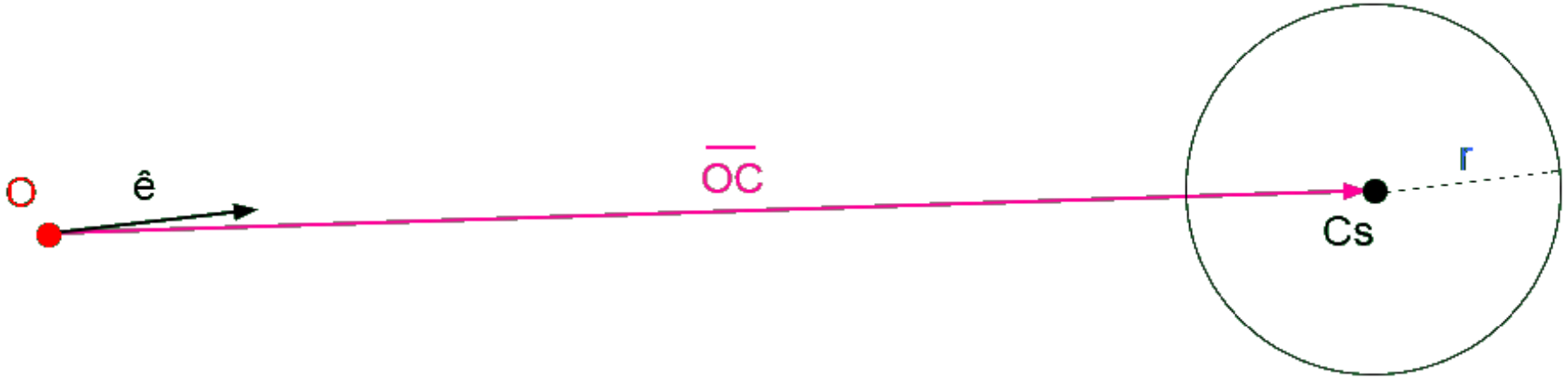
```
# Sphere
Cs = np.array([2, 0])          # Center of sphere
r = 1.5                        # Radius of sphere
```





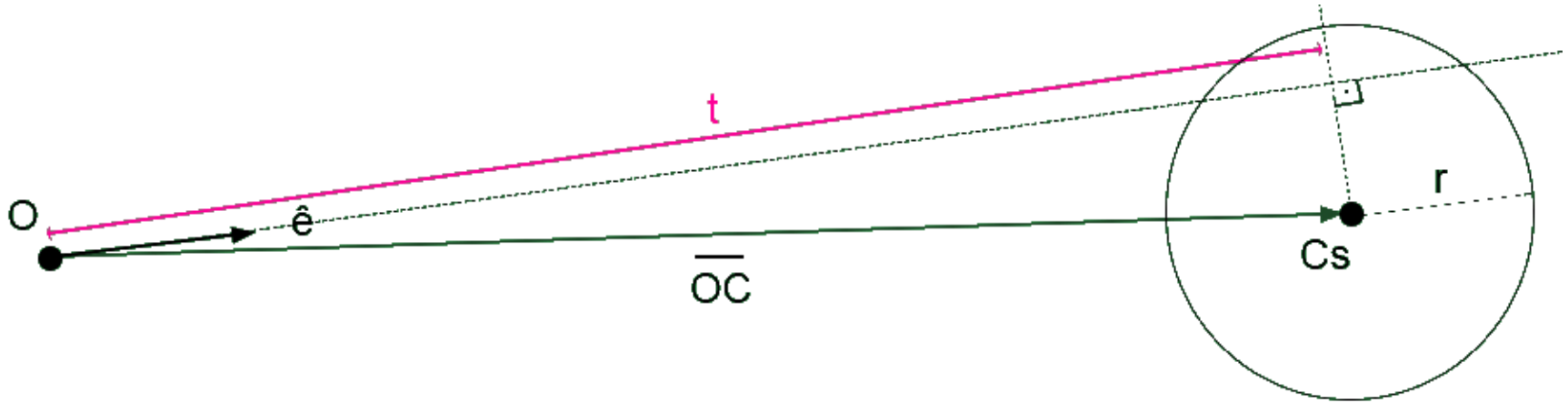
Ray-Object Intersections: Primary rays

```
# Sphere
OC_ = Cs-O  # Oriented segment from origin to center of the sphere
```



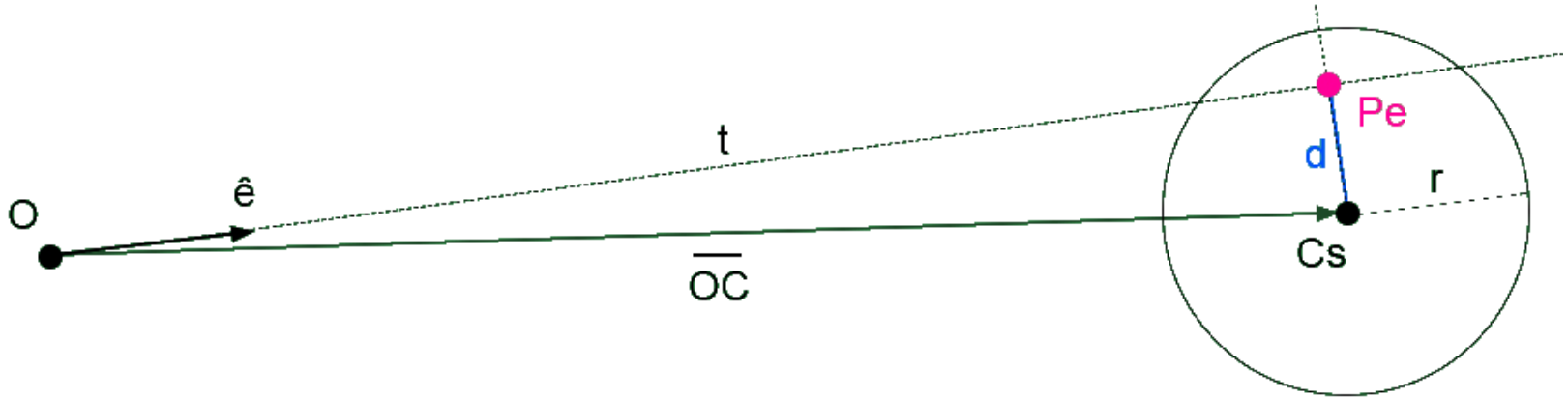
Ray-Object Intersections: Primary rays

```
# find the parameter t which will parametrize the ray segment and
# find the intersections from the origin

t = np.dot(OC_, e_)    # Vector projection of OC_ on e_
```



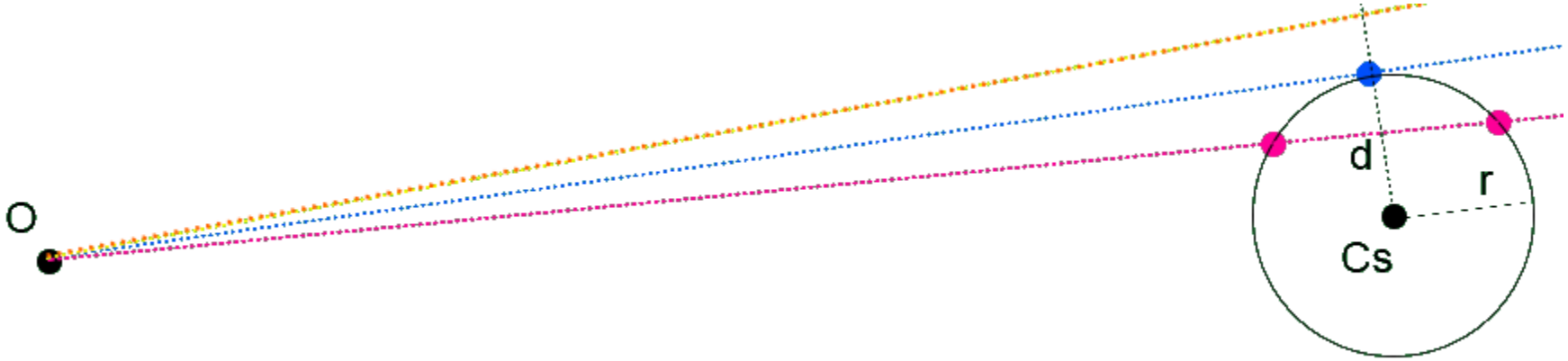
Ray-Object Intersections: Primary rays

```
Pe = O + e_*t                    # Point on vector e_ projected from OC_
d = np.linalg.norm(Pe - Cs)      # Distance from the point Pe and the center
                                 # of the sphere
```



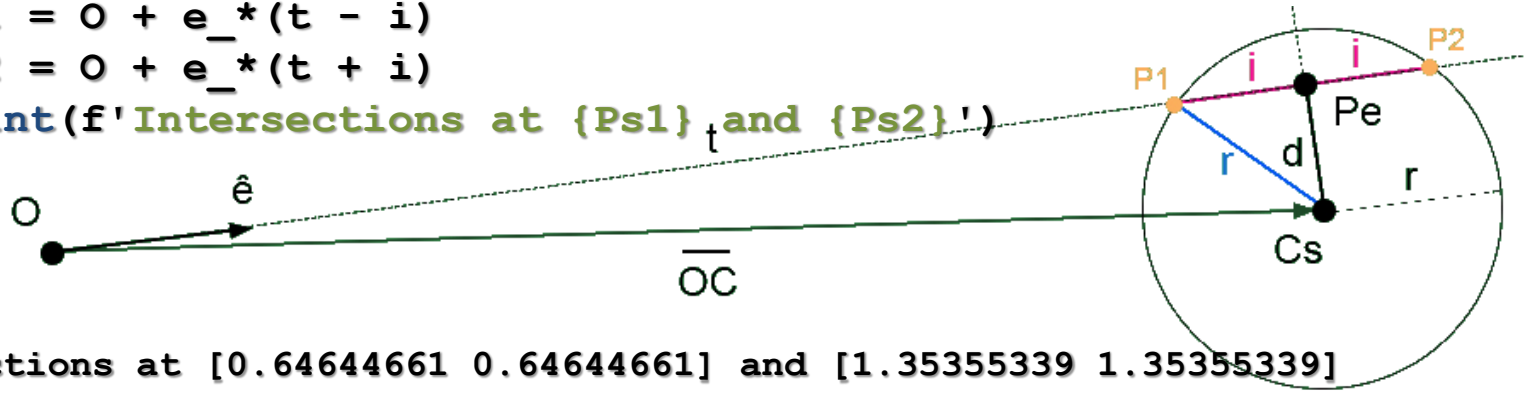
Ray-Object Intersections: Primary rays

If d>r then there is no intersections
If d=r then there is 1 intersection (tangent)
if d<r there are 2 intersections.



Ray-Object Intersections: Primary rays

```
# Position of the intersections
if(d > r):
    print("No intersection!")
elif(d == r):
    Ps = Pe
    print(f'Intersection at {Ps}')
else:
    i = (r**2 - d**2)**0.5
    Ps1 = O + e_*(t - i)
    Ps2 = O + e_*(t + i)
    print(f'Intersections at {Ps1} and {Ps2}')
```




```
Intersections at [0.64644661 0.64644661] and [1.35355339 1.35355339]
```

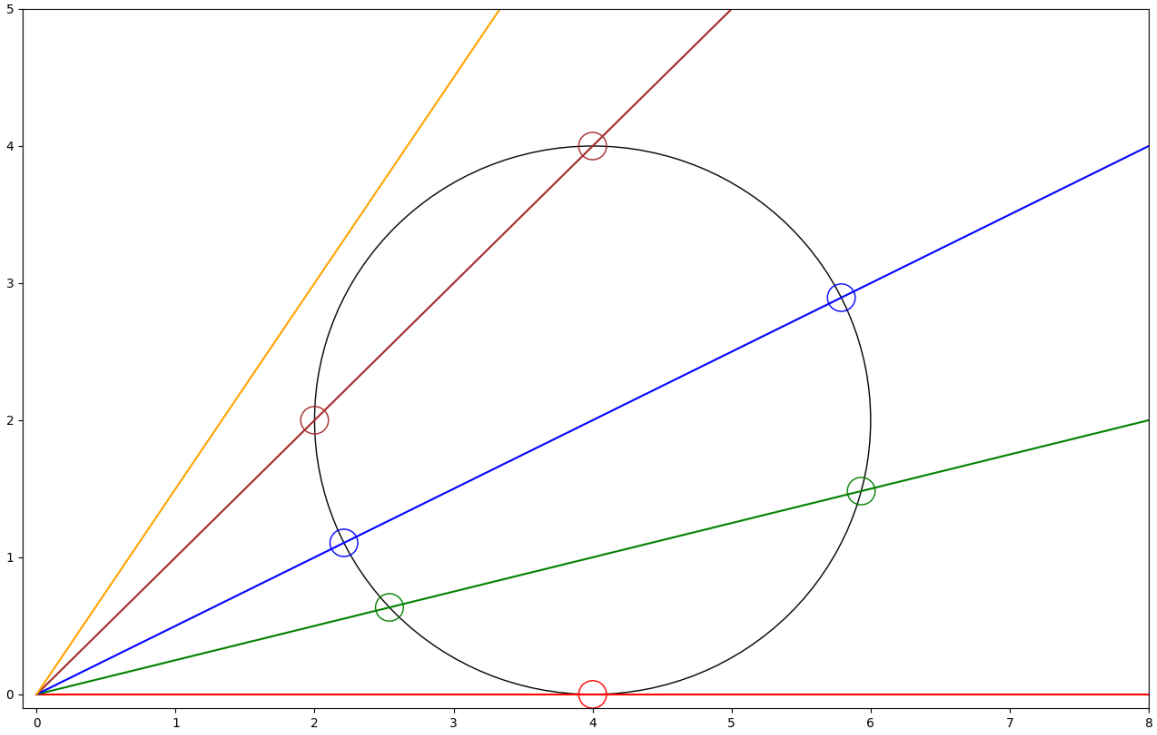
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
import matplotlib.pyplot as plt
import numpy as np

# Define x points to draw lines
x = np.array([0, 8])
colors = ["red", "green", "blue", "brown","orange"]

# Define angular coeficients
M = [0, 0.25, 0.5, 1, 1.5]

# Define sphere
Cs = [4, 2]
r = 2
Circle = plt.Circle(Cs, r, color="k", fill=0)
ax.add_artist(Circle)
```

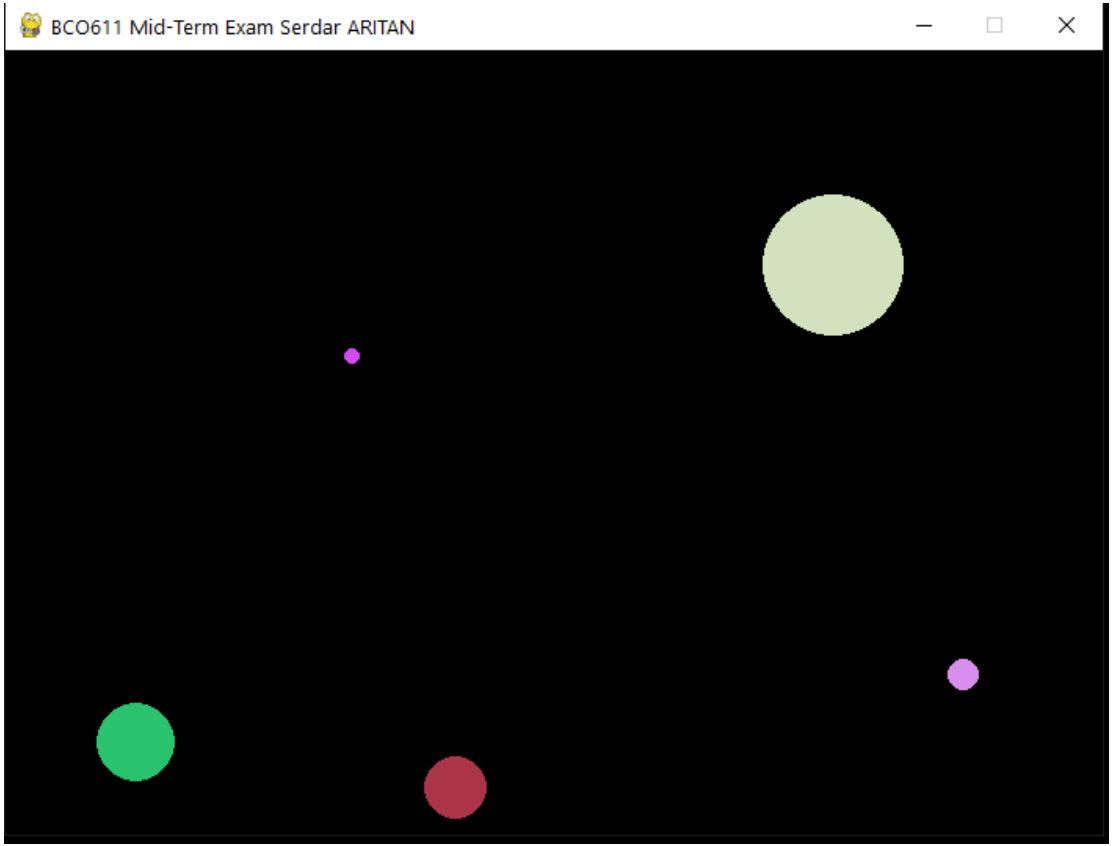
```
# Draw intersections
for index, m in enumerate(M):
    y = m*x
    plt.plot(x, y, color=colors[index])
    plt.xlim([-0.1, 8])
    plt.ylim([-0.1, 5])

    # Define ray
    O = np.array([0, 0])
    e_ = np.array([1, m])
    e_ = e_/np.linalg.norm(e_)

    # Intersection
    OC_ = Cs - O
    t = np.dot(OC_, e_)
    Pe = O + e_*t
    d = np.linalg.norm(Pe - Cs)
```


```
    # Draw intersections
    if(d == r):
        Ps = Pe
        Circle =plt.Circle(Ps, 0.1, color=colors[index], fill=0)
        ax.add_artist(Circle)
    if(d < r):
        i = (r**2 - d**2)**0.5
        Ps1 = O + e_*(t - i)
        Circle =plt.Circle(Ps1, 0.1, color=colors[index], fill=0)
        ax.add_artist(Circle)
        Ps2 = O + e_*(t + i)
        Circle =plt.Circle(Ps2, 0.1, color=colors[index], fill=0)
        ax.add_artist(Circle)

plt.show()
```

NumPy



PHYSICS in COMPUTER ANIMATIONS and GAMES



Mid-Term Exam

Using Ray-Tracing Technique is Optional but it is worth it

- Write a **`collision detection`** function in python.
  - When the collection occurs function returns **the angle and the point of collision**
- Write a **`post-collision velocity calculation`** function.
  - Function returns the post-collision velocities of the bodies as vector values.
  - Input arguments of the function is **the angle and the point of collision, masses of the bodies, the coefficient of restitution**

YOUR PROGRAM MUST BE RETURN

NO LATER THAN 18.30 THE TUESDAY, 2nd April 2024

Email To: serdar.aritan@hacettepe.edu.tr and serdar.aritan@gmail.com

Subject: BCO611 Mid-Term <Student No>