Ray Tracing

#7

Serdar ARITAN

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

How Ray Tracing Works

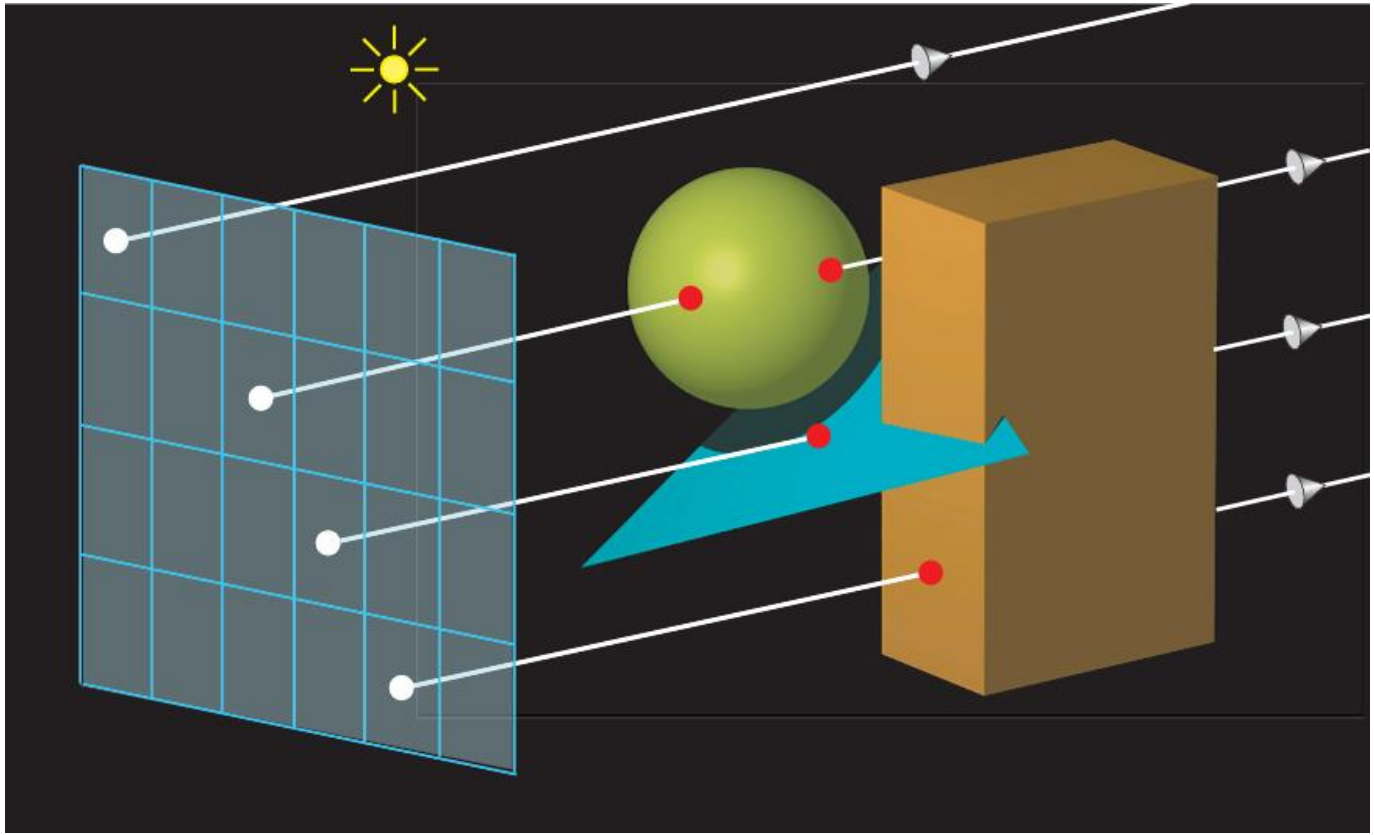
A simple ray tracer works by performing the following operations:

```
define some objects
specify a material for each object
define some light sources
define a window whose surface is covered with pixels
for each pixel
    shoot a ray towards the objects from the center of the pixel
    compute the nearest hit point of the ray with the objects (if any)

    if the ray hits an object
        use the object's material and the lights to compute the pixel color
    else
        set the pixel color to black
```

How Ray Tracing Works

The pixels are on a plane called the view plane, which is perpendicular to the rays.

The rays are parallel to each other and produce an orthographic projection of the objects.



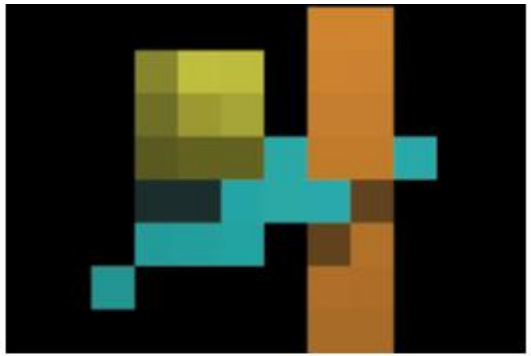
How Ray Tracing Works

When **a ray** hits an **object**, the color of its pixel is computed from the way the object's material reflects light, a process that's known as **shading**. Although the pixels on the view plane are just mathematical abstractions, like everything else in the ray tracer, each one is associated with a real pixel in a window on a computer screen.

The process of working out where a ray hits an object is known as **the ray object intersection calculation**. This is a fundamental process in ray tracing and usually takes most of the time. The intersection calculation is different for each type of object; some objects are easy to intersect, while others are difficult. All intersection calculations require some mathematics.

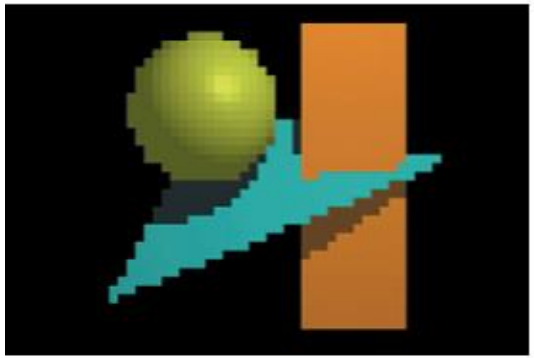
How Ray Tracing Works

6x4

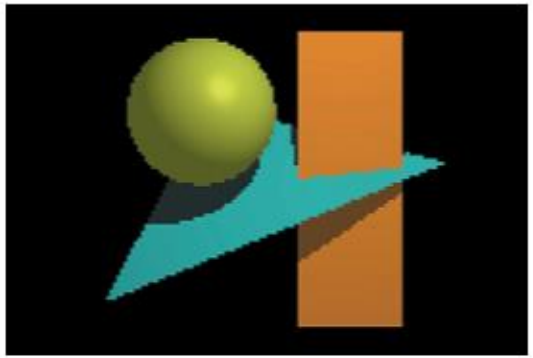


12x8

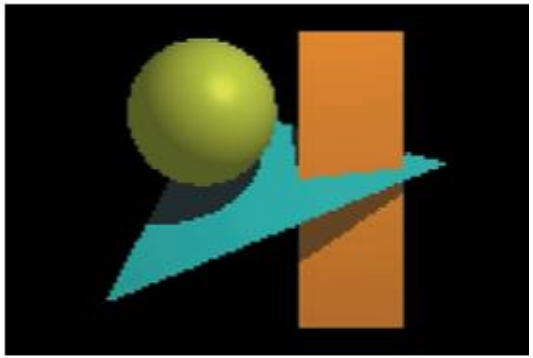
24x16



60x40



120x80



150x100

How Ray Tracing Works

The World

Ray tracers render scenes that contain the geometric objects, lights, a camera, a view plane, a tracer, and a background color. In this course, the world will only store the objects and view plane.

The locations and orientations of all scene elements are specified in world coordinates, which is a 3D Cartesian coordinate system.

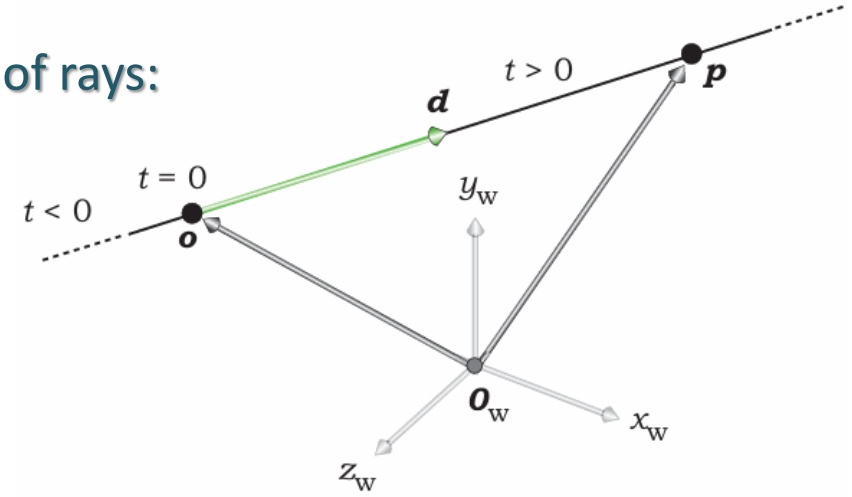
How Ray Tracing Works

Rays

A ray is an infinite straight line that's defined by a point o, called the origin, and a unit vector d, called the direction. A ray is parametrized with the ray parameter t, where t = 0 at the ray origin, so that an arbitrary point p on a ray can be expressed as ***p = o + td***.

Ray tracing uses the following types of rays:

- primary rays;
- secondary rays;
- shadow rays;
- light rays.



How Ray Tracing Works

Primary rays start at the centers of the pixels for parallel viewing, and at the camera location for perspective viewing.

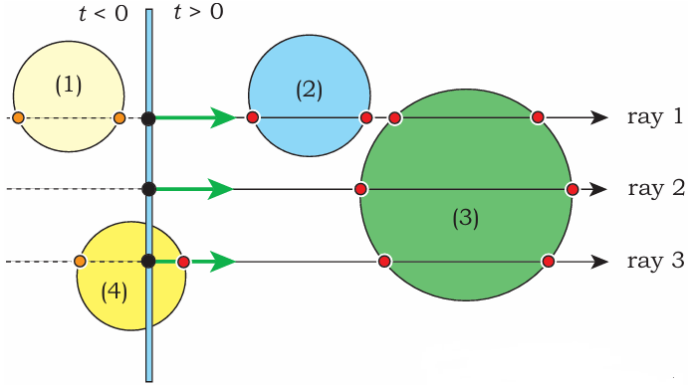
Secondary rays are reflected and transmitted rays that start on object surfaces.

Shadow rays are used for shading and start at object surfaces.

Light rays start at the lights and are used to simulate certain aspects of global illumination, such as caustics.

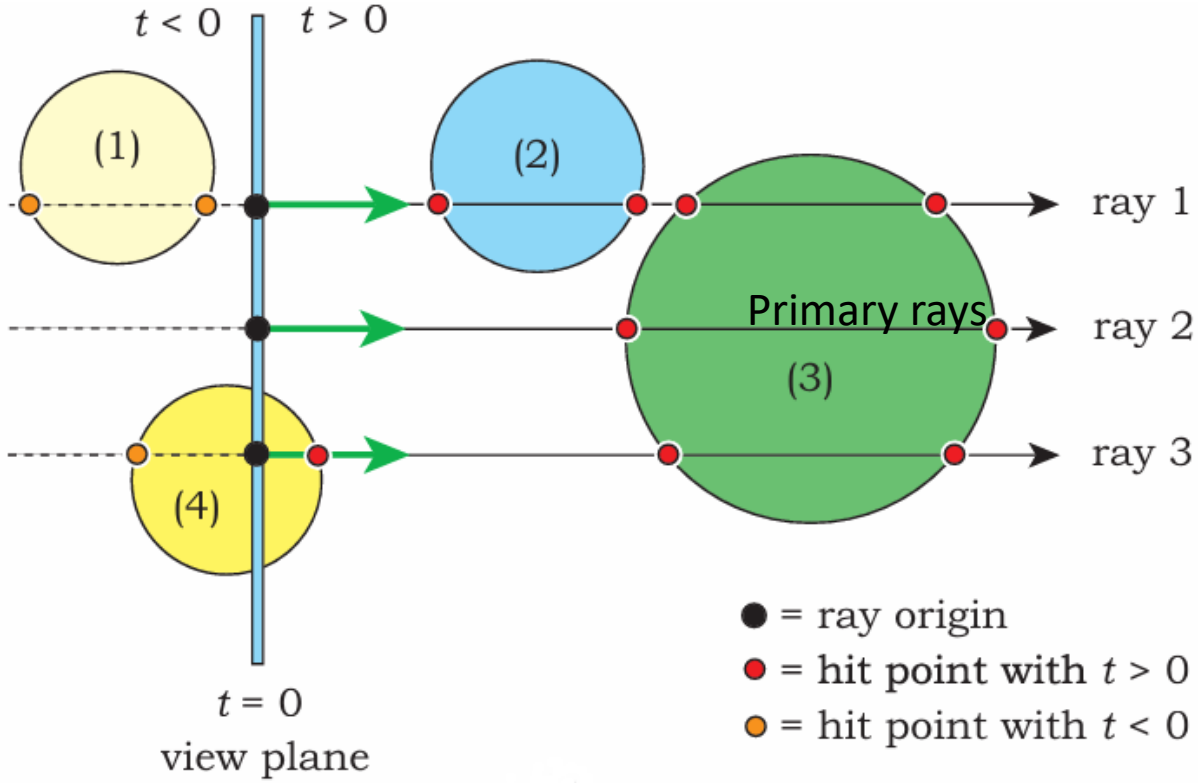
Ray-Object Intersections

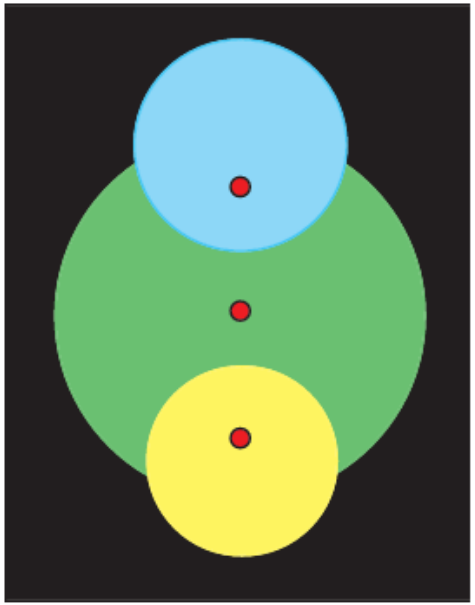
The basic operation we perform with a ray is to intersect it with all geometric objects in the scene. This finds the nearest hit point, if any, along the ray from **o** in the direction **d**.



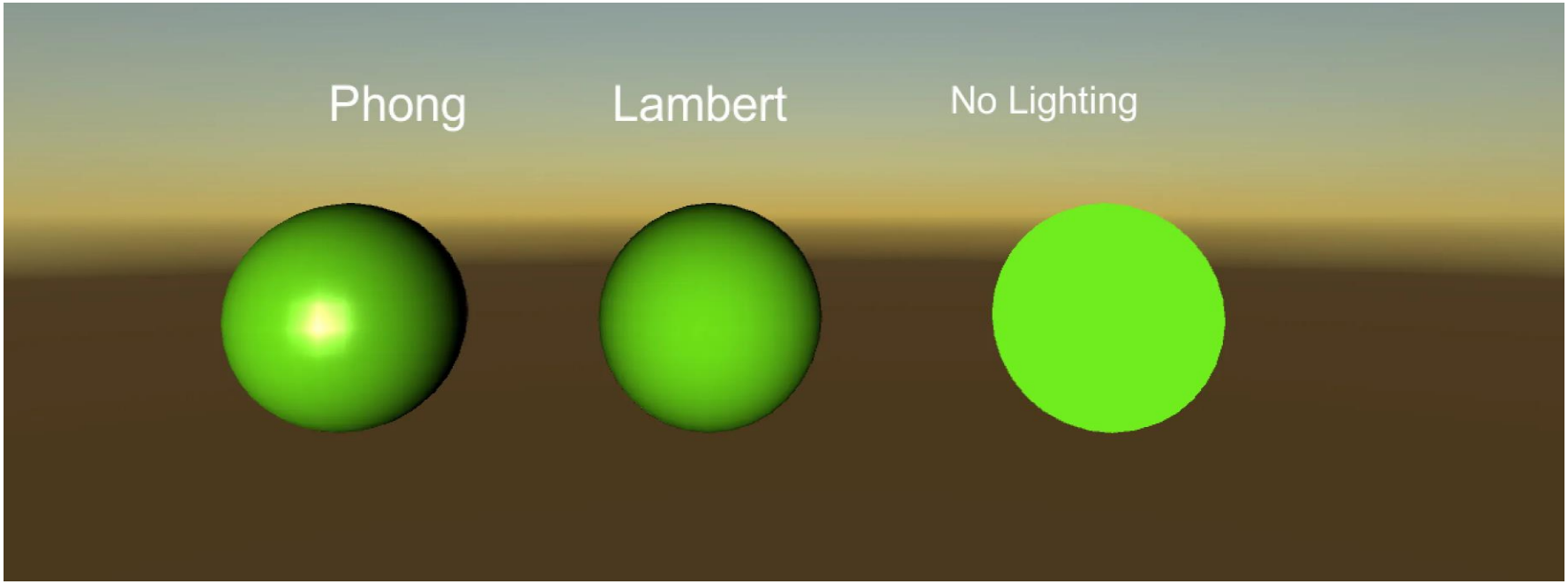
Sphere (1) is behind the origin of all rays that intersect it (t < 0) and will not appear in the image.
Sphere (2) will be rendered with ray 1 and with all rays that hit it.
Sphere (3) will only be rendered with rays like ray 2 that don't hit any other spheres.
Sphere (4) will only be rendered with rays like ray 3 that start inside it

Ray-Object Intersections: Primary rays

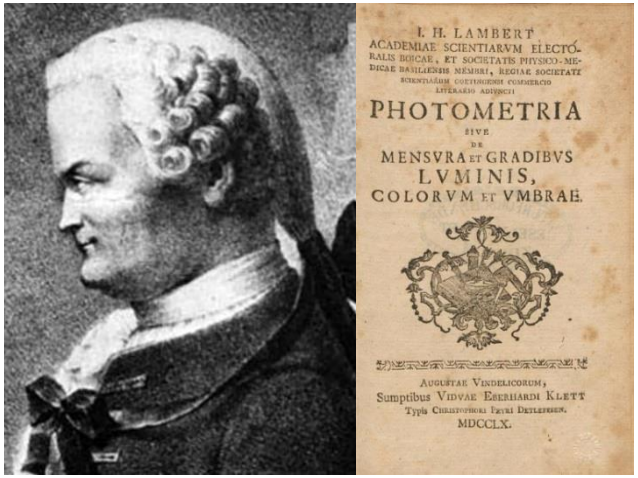


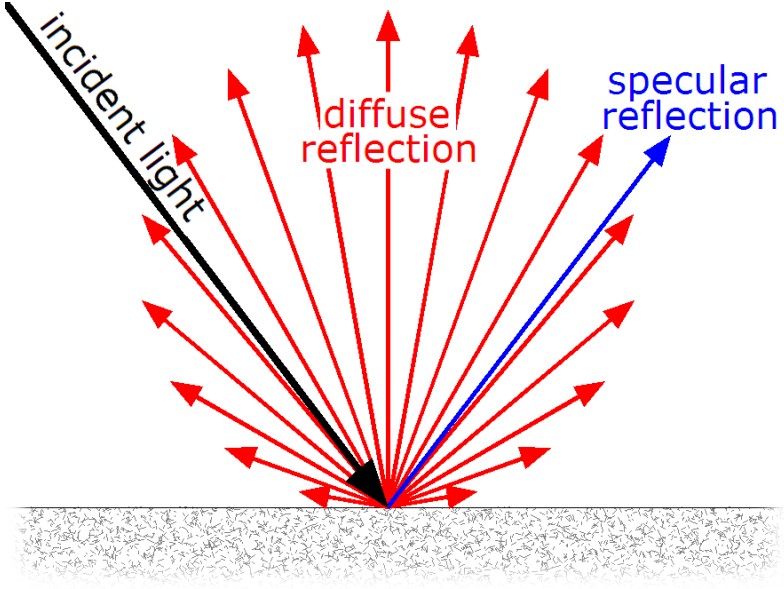


PHONG - LAMBERT - NO LIGHTING



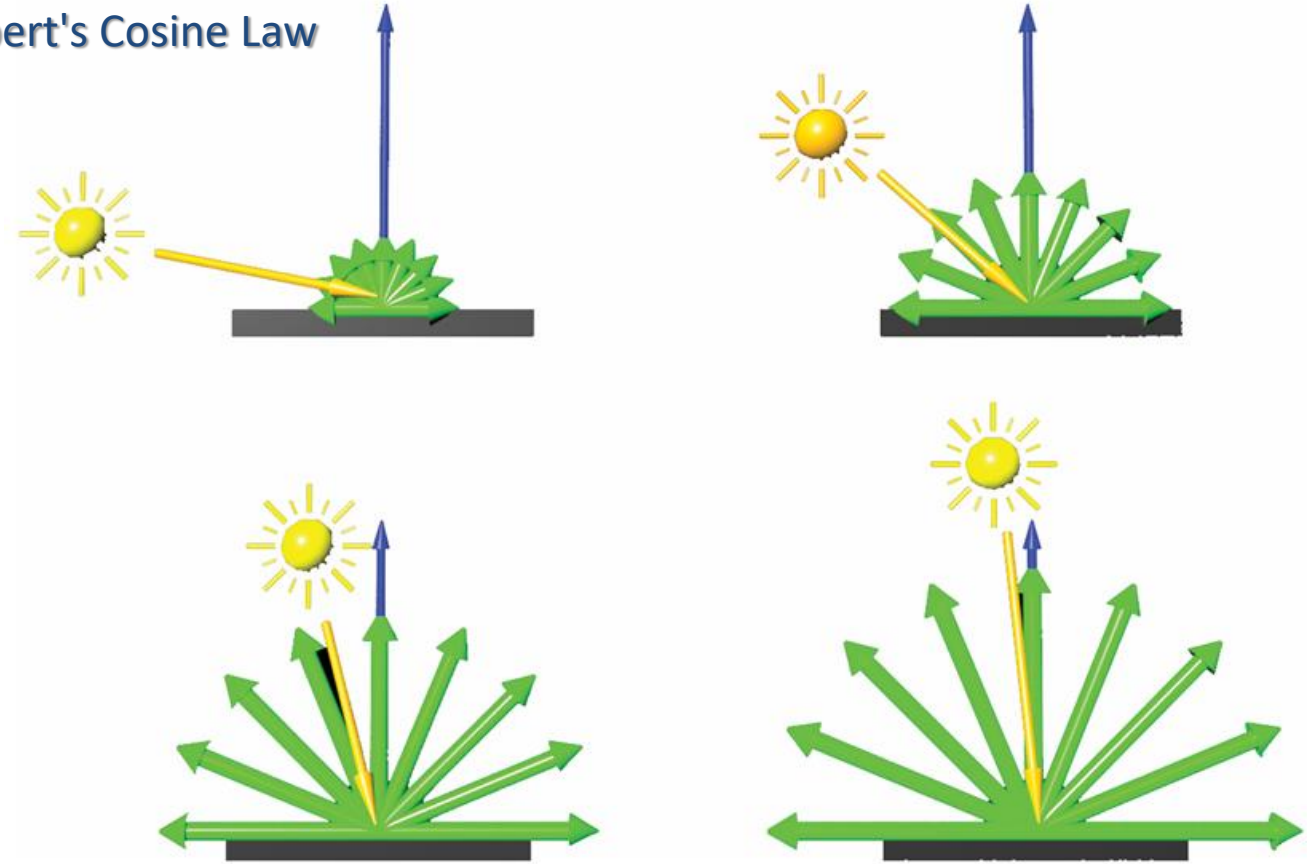
Lambert Lighting



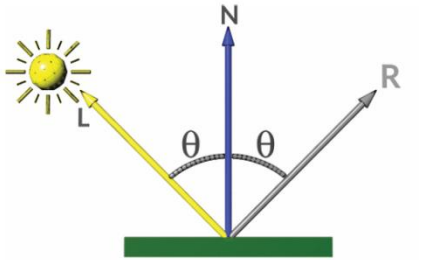


Diffuse and specular reflection from a glossy surface. The rays represent luminous intensity, which varies according to Lambert's cosine law for an ideal diffuse reflector.

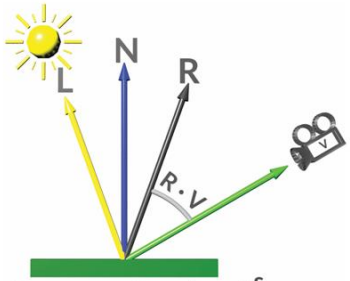
Lambert's Cosine Law



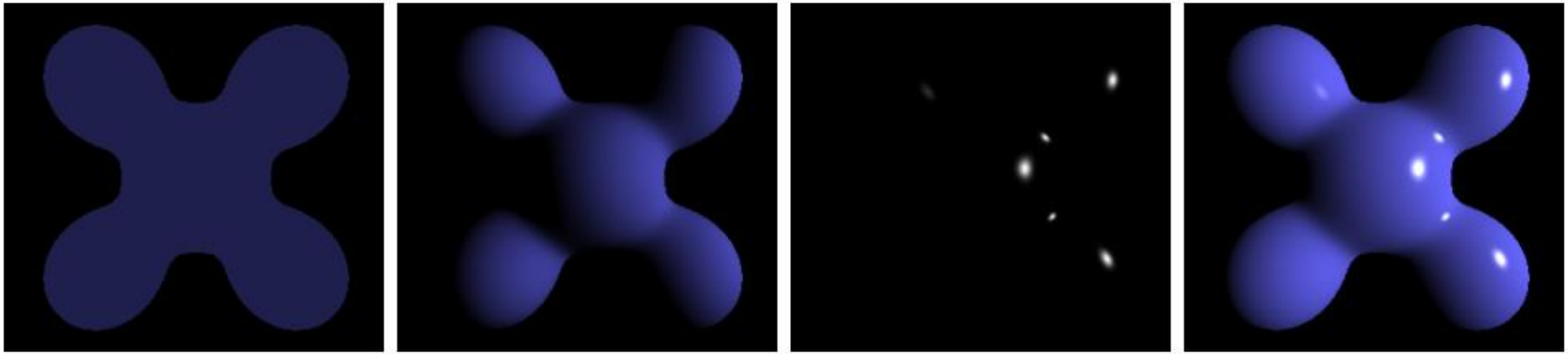
The ambient component of light is simply an offset from black generated by the shader.



Angle of Incidence = Reflectance



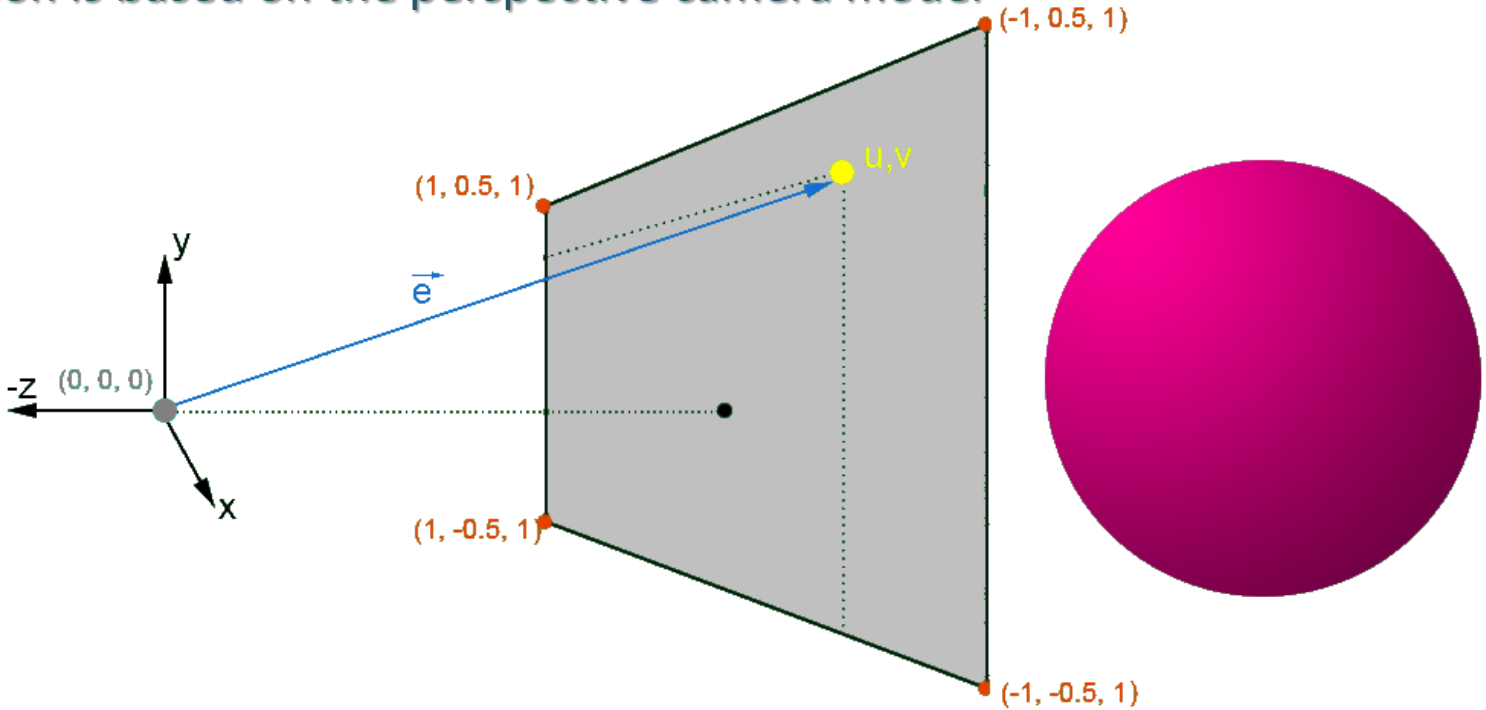
Phong Specular = $(R \cdot V)^s$



Ambient + Diffuse + Specular = Phong Reflection

3D Intersection

Apply same model for each pixel of an image plane as the origin and the ray direction is based on the perspective camera model



3D Intersection

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

N, M = 256j, 512j
O = np.ones((int(N.imag), int(M.imag), 3))                  # Init image plane origin
O[..., 1], O[..., 0] = np.mgrid[0.5:-0.5:N, 1:-1:M]        # Image plane uvw coordinates
e_ = O/np.linalg.norm(O, axis=2)[:,:,np.newaxis]            # Normalized ray directon e_

# Sphere
Cs = np.array([0, 0, 4])                                    # Center of sphere
r = 1.5                                                     # Radius of sphere

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

vec_dot = np.vectorize(np.dot, signature='(n),(m)->()')# Vectorize dot product function
t = vec_dot(OC_, e_)                                        # Pixelwise dot product
Pe = O + e_*t[:,:,np.newaxis]          # Point on vector e_ projected from OC_
d = np.linalg.norm(Pe - Cs, axis=2)# Distance from the point Pe and the center of the
sphere
```

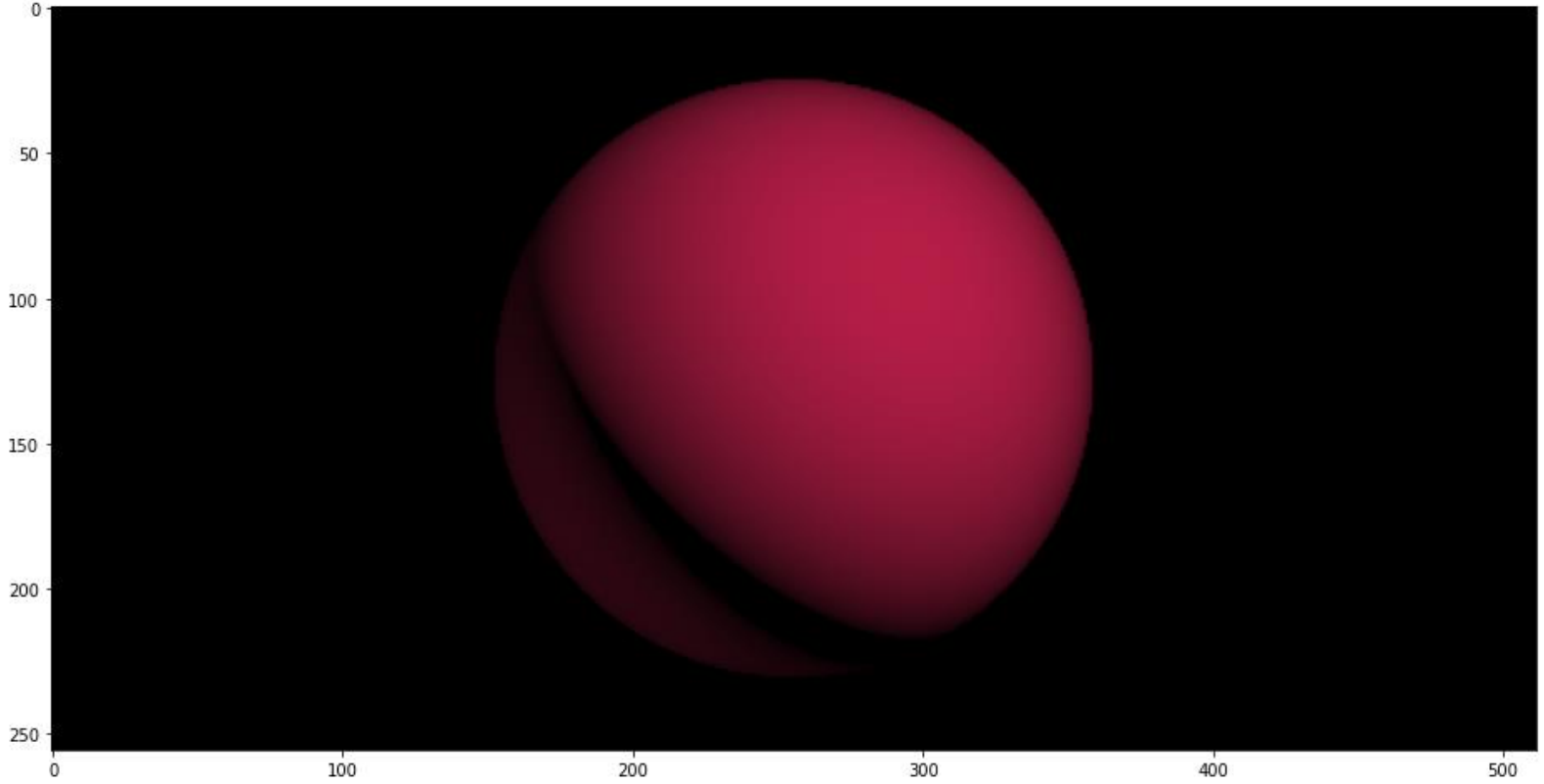
3D Intersection

```
# Find intersection position
i = (r**2 - d**2)**0.5
Ps = O + e_*(t - i)[:,:,np.newaxis]

# Facing ratio (incidence value)
i_ = i[:,:,np.newaxis]/r

# Calculate the normal vector for each point
n = Ps - Cs                                                    # Calculate vector n
n_ = n/np.linalg.norm(n, axis=2)[:,:,np.newaxis]               # Normalize n

# Simple directional light model
Cd = np.array([0.9, 0.15, 0.35])**(1/0.455) # Sphere diffuse color with gamma
# Key light
l = np.array([-1.5, 1.5, -1])                                  # Key light vector
l_ = l/np.linalg.norm(l)                                       # Key light vector normalization
Kd = vec_dot(l_, n_)[:,:,np.newaxis]                           # Calculate light incidence
Kd[Kd < 0] = 0                                                 # clamp negative values
diff = Cd*Kd                                                   # Writes to diffuse
```

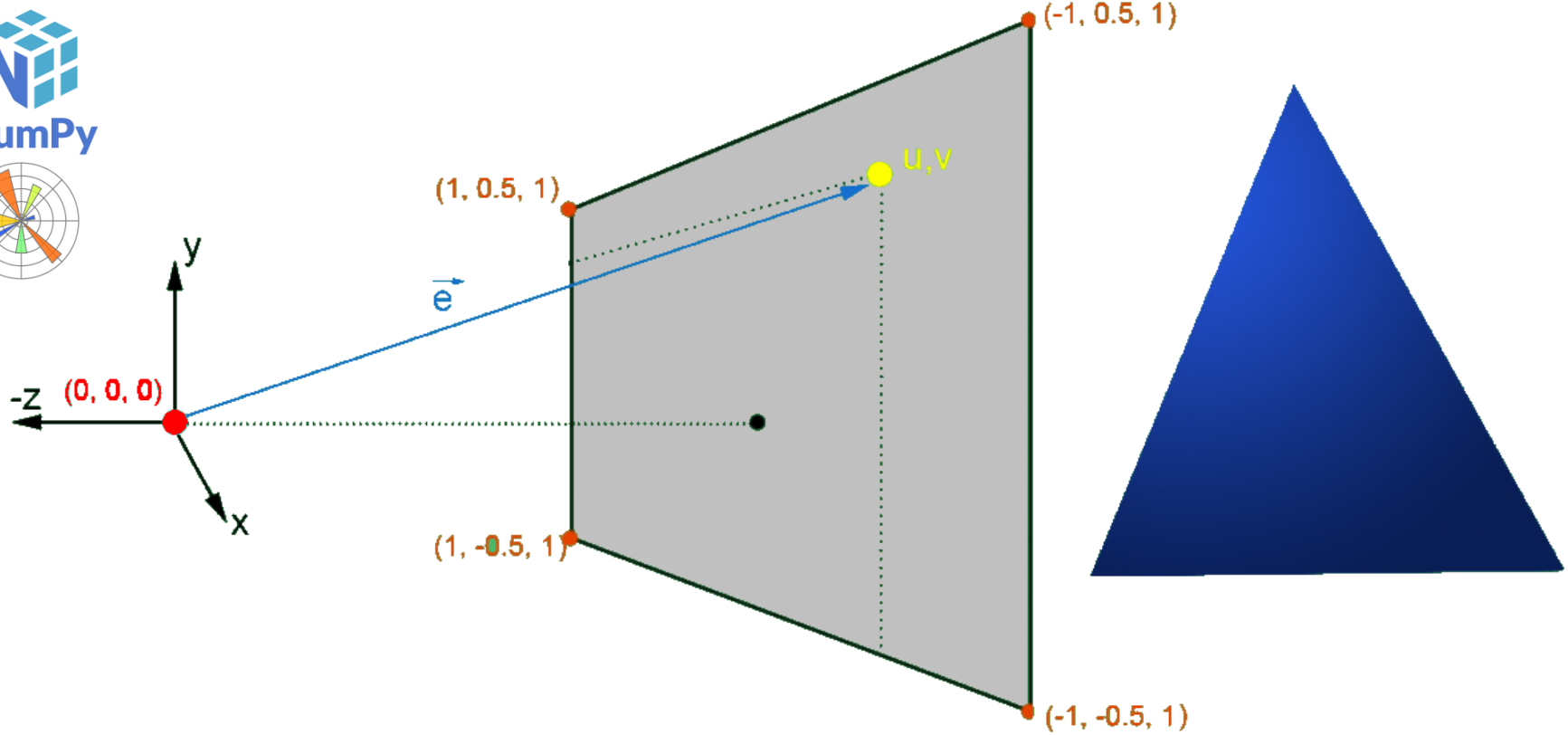
3D Intersection


```
# Back light
l = np.array([1.5, -1, 1])                          # Back light vector
l_ = l/np.linalg.norm(l)                            # Back light vector normalization
Kd = vec_dot(l_, n_)[:,:,np.newaxis]                # Calculate light incidence
Kd[Kd < 0] = 0                                      # clamp negative values
diff += Cd*Kd*0.25                                  # Adds to diffuse

output = np.zeros((int(N.imag), int(M.imag), 3))    # Init output image
output[d < r] = (diff*i_)[d < r]                    # Shades diffuse and fr

output[output < 0] = 0; output[output > 1] = 1      # Clamp values before visualization

# Visualization
fig, ax = plt.subplots(figsize=(16, 10))
ax.imshow(output**(1/2.2))      # View the resulting image with gamma adjustment (sRGB)
plt.show()
```

PHYSICS in COMPUTER ANIMATIONS and GAMES



NumPy



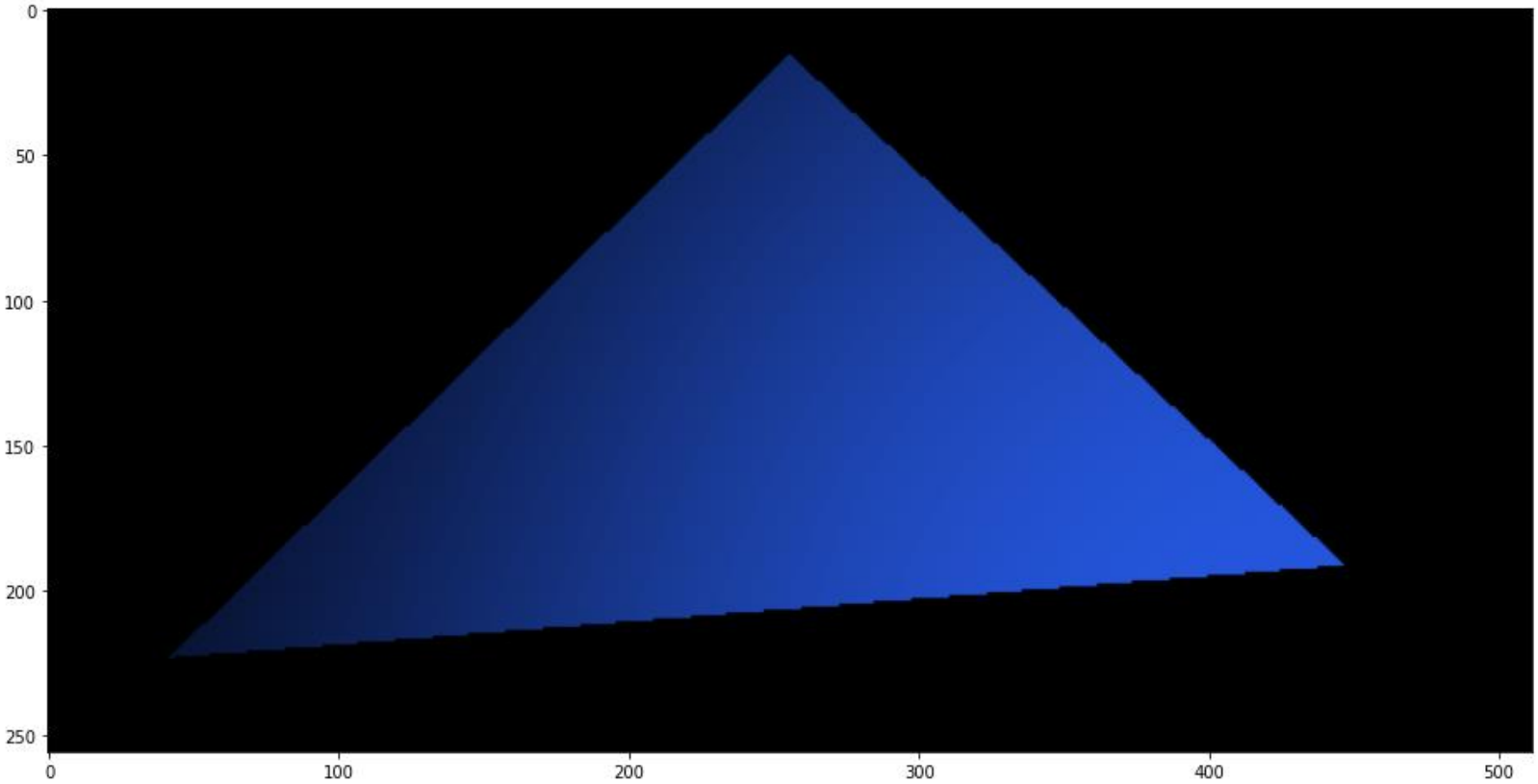
PHYSICS in COMPUTER ANIMATIONS and GAMES

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

N, M = 256j, 512j
O = np.ones((int(N.imag), int(M.imag), 3))                # Init image plane origin
O[..., 1], O[..., 0] = np.mgrid[0.5:-0.5:N, 1:-1:M]       # Image plane uvw coordinates
e_ = O/np.linalg.norm(O, axis=2)[:,:,np.newaxis]          # Normalized ray directon e_

# Triangle
A = np.array([0   , 2.2 , 5])                                  # Point A
B = np.array([6.7 , -3  , 8])                                  # Point B
C = np.array([-1.5, -0.5, 2])                                  # Point C

AB = B - A                                                     # Oriented segment A to B
AC = C - A                                                     # Oriented segment A to C
n = np.cross(AB, AC)                                           # Normal vector
n_ = n/np.linalg.norm(n)                                       # Normalized normal

# Using the point A to find d
d = - np.dot(n_, A)
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

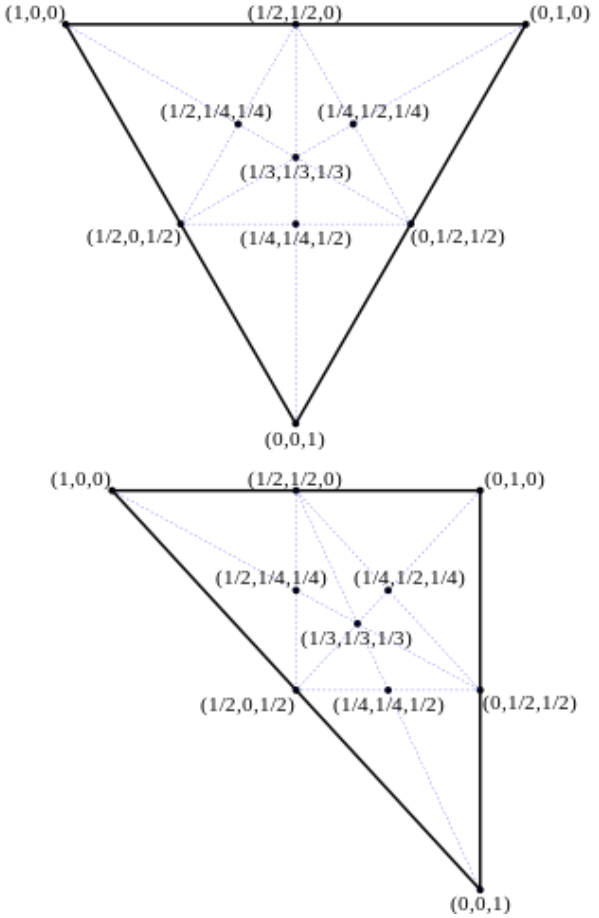
```
# Finding parameter t
vec_dot = np.vectorize(np.dot, signature='(n),(m)->()') # Vectorize dot product
function
t = - (vec_dot(n_, O) + d)/vec_dot(n_, e_)              # Get t for each pixel

# Finding P
P = O + t[..., np.newaxis]*e_

# Get the resulting vector for each vertex
# following the construction order
Pa = vec_dot(np.cross(B - A, P - A), n_)                # Resulting vector of A
Pb = vec_dot(np.cross(C - B, P - B), n_)                # Resulting vector of B
Pc = vec_dot(np.cross(A - C, P - C), n_)                # Resulting vector of C

output = np.zeros((int(N.imag), int(M.imag), 3))        # Init output image
# Inside the triangle conditionals
cond = np.logical_and(np.logical_and(Pa >= 0, Pb >= 0), Pc >= 0)
fr = vec_dot(n_, -e_)[..., np.newaxis]                  # Compute the facing ratio
output[cond] = (0.15, 0.35, 0.9)*fr[cond]               # Shade with color and fr
# Visualization
fig, ax = plt.subplots(figsize=(16, 10))
ax.imshow(output)
plt.show()
```

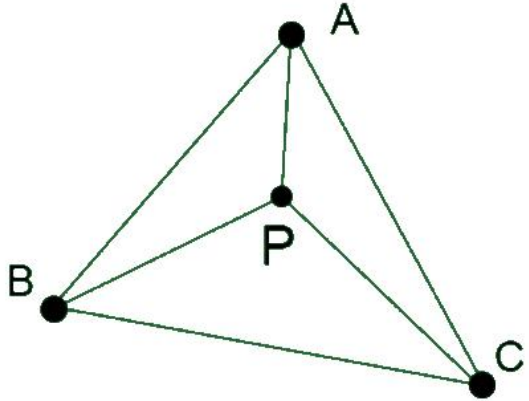
PHYSICS in COMPUTER ANIMATIONS and GAMES

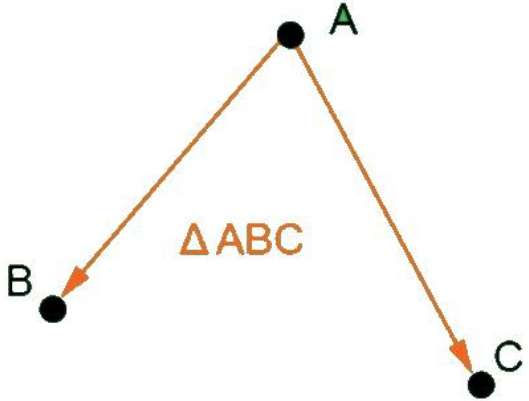


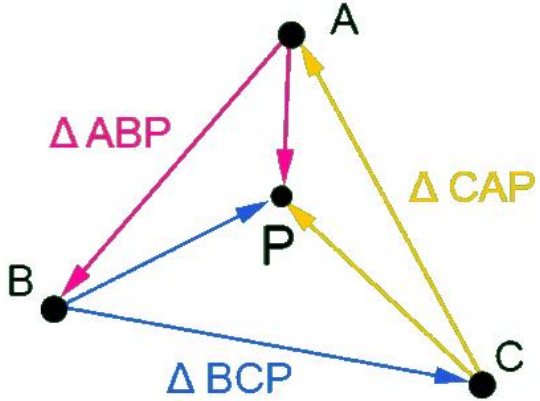
Barycentric coordinates

The barycentric coordinates of a point can be interpreted as **masses** placed at the **vertices** of the simplex, such that the point is the **center of mass** (or barycenter) of these masses. These masses can be zero or negative; they are all positive if and only if the point is inside the simplex. Generalized barycentric coordinates have applications in computer graphics and more specifically in geometric modelling.









$$AREA_{ABC} = \frac{\|(B-A)\ \mathrm{x}\ (C-A)\|}{2}$$

$$\propto = \frac{AREA_{BCP}}{AREA_{ABC}} = \frac{\|(C-B)\ \mathrm{x}\ (P-B)\|}{\|(B-A)\ \mathrm{x}\ (C-A)\|}$$

$$\beta = \frac{AREA_{CAP}}{AREA_{ABC}} = \frac{\|(A-C)\ \mathrm{x}\ (P-C)\|}{\|(B-A)\ \mathrm{x}\ (C-A)\|}$$

$$\gamma = \frac{AREA_{ABP}}{AREA_{ABC}} = \frac{\|(B-A)\ \mathrm{x}\ (P-A)\|}{\|(B-A)\ \mathrm{x}\ (C-A)\|}$$

PHYSICS in COMPUTER ANIMATIONS and GAMES

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

N, M = 256j, 512j
O = np.ones((int(N.imag), int(M.imag), 3))                # Init image plane origin
O[..., 1], O[..., 0] = np.mgrid[0.5:-0.5:N, 1:-1:M]       # Image plane uvw coordinates
e_ = O/np.linalg.norm(O, axis=2)[:,:,np.newaxis]          # Normalized ray directon e_

# Triangle
A = np.array([0 , 1.25 , 3])                              # Point A
B = np.array([2 , -1.25, 3])                              # Point B
C = np.array([-2, -1.25, 3])                              # Point C

AB = B - A                                                   # Oriented segment A to B
AC = C - A                                                   # Oriented segment A to C
n = np.cross(AB, AC)                                         # Normal vector
n_ = n/np.linalg.norm(n)                                     # Normalized normal

# Using the point A to find d
d = - np.dot(n_, A)
```

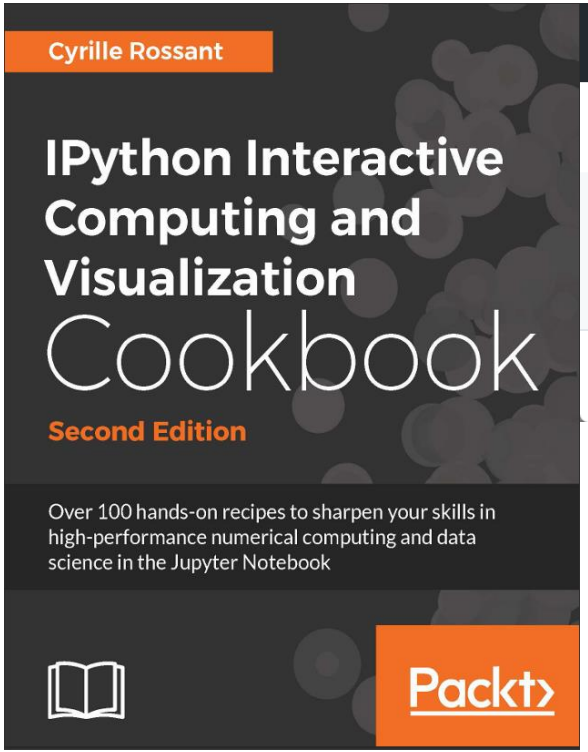
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Finding parameter t
vec_dot = np.vectorize(np.dot, signature='(n),(m)->()') # Vectorize dot product
function
t = - (vec_dot(n_, O) + d)/vec_dot(n_, e_)              # Get t for each pixel

# Finding P
P = O + t[..., np.newaxis]*e_

# Get the resulting vector for each vertex
# following the construction order
Pa = vec_dot(np.cross(B - A, P - A), n_)          # Resulting vector of A
Pb = vec_dot(np.cross(C - B, P - B), n_)          # Resulting vector of B
Pc = vec_dot(np.cross(A - C, P - C), n_)          # Resulting vector of C
cond = np.logical_and(np.logical_and(Pa >= 0, Pb >= 0), Pc >= 0)
# Calculate barycentric coordinates
Aa = np.cross(B - A, P - A)                       # Resulting vector of A and P
Aa = np.linalg.norm(Aa, axis=2)                   # Area of triangle ABP
Ab = np.cross(C - B, P - B)                       # Resulting vector of B and P
Ab = np.linalg.norm(Ab, axis=2)                   # Area of triangle BCP
Ac = np.cross(A - C, P - C)                       # Resulting vector of C and P
Ac = np.linalg.norm(Ac, axis=2)                   # Area of triangle CAP
At = np.cross(C - A, B - A)                       # Resulting vector of triangle
At = np.linalg.norm(At)                           # Area of triangle ABC
```

```
# Getting the barycenter weights
alpha = (Ab/At)[..., np.newaxis]
beta = (Ac/At)[..., np.newaxis]
gamma = (Aa/At)[..., np.newaxis]

# Output image
output = np.zeros((int(N.imag), int(M.imag), 3))                # Init output image
Ca = np.array([1, 0, 0.4])                                      # Color vertex A
Cb = np.array([0.4, 1, 0])                                      # Color vertex B
Cc = np.array([0, 0.4, 1])                                      # Color vertex C
# Interpolated color based on barycentric coordinates
Cd = (alpha*Ca + beta*Cb + gamma*Cc)/(alpha + beta + gamma)

output[cond] = Cd[cond]                          # Shade with the interpolated colors

# Visualization
fig, ax = plt.subplots(figsize=(16, 10))
ax.imshow(output)
plt.show()# Area of triangle ABC
```


0
50
100
150
200
250
0
100
200
300
400
500

PHYSICS in COMPUTER ANIMATIONS and GAMES



GitHub Gist Search... All gists Back to GitHub Sign in Sign up

Instantly share code, notes, and snippets.

rossant / raytracing.py

Last active 3 months ago

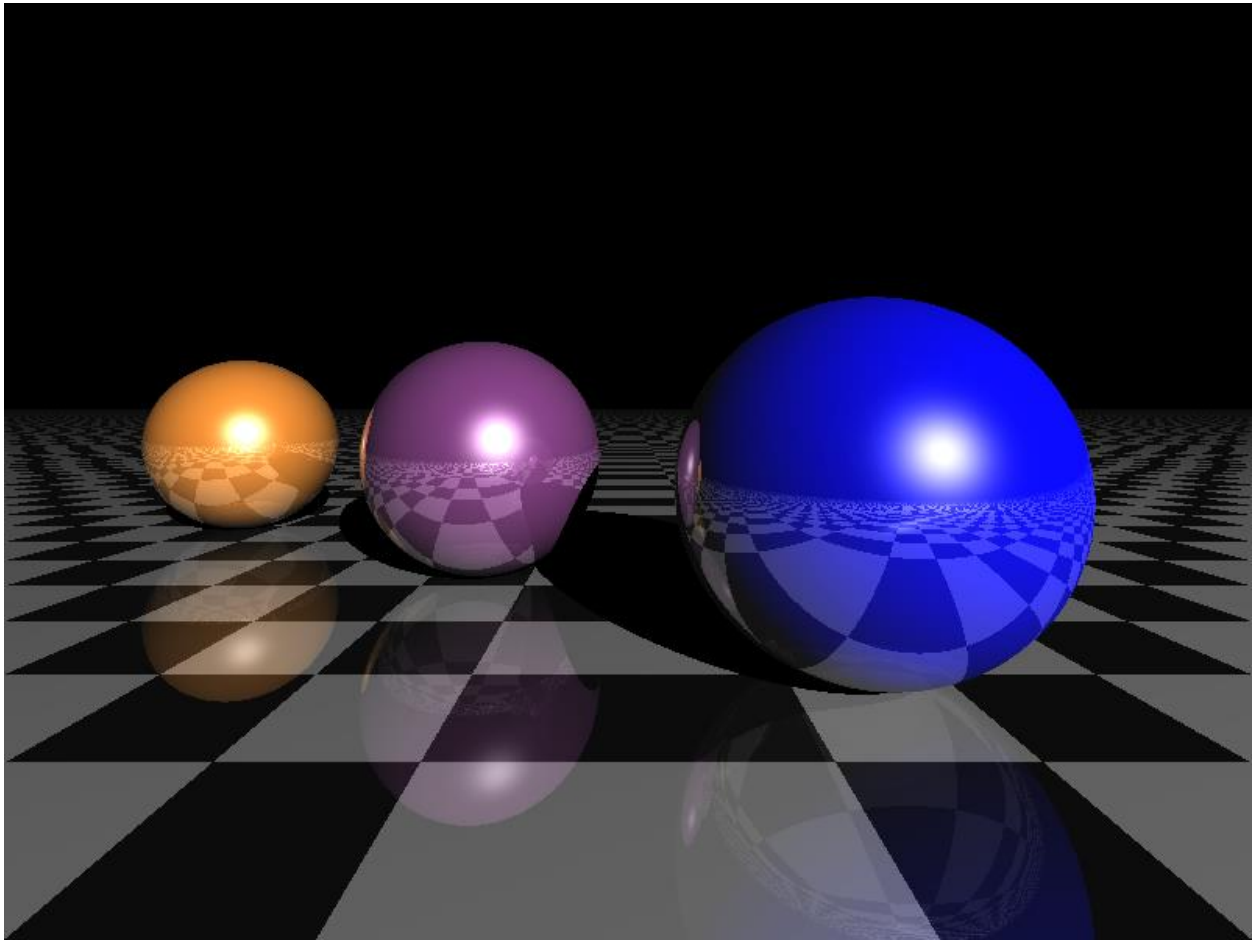
Star 164 Fork 56

Code Revisions 6 Stars 164 Forks 56 Download ZIP

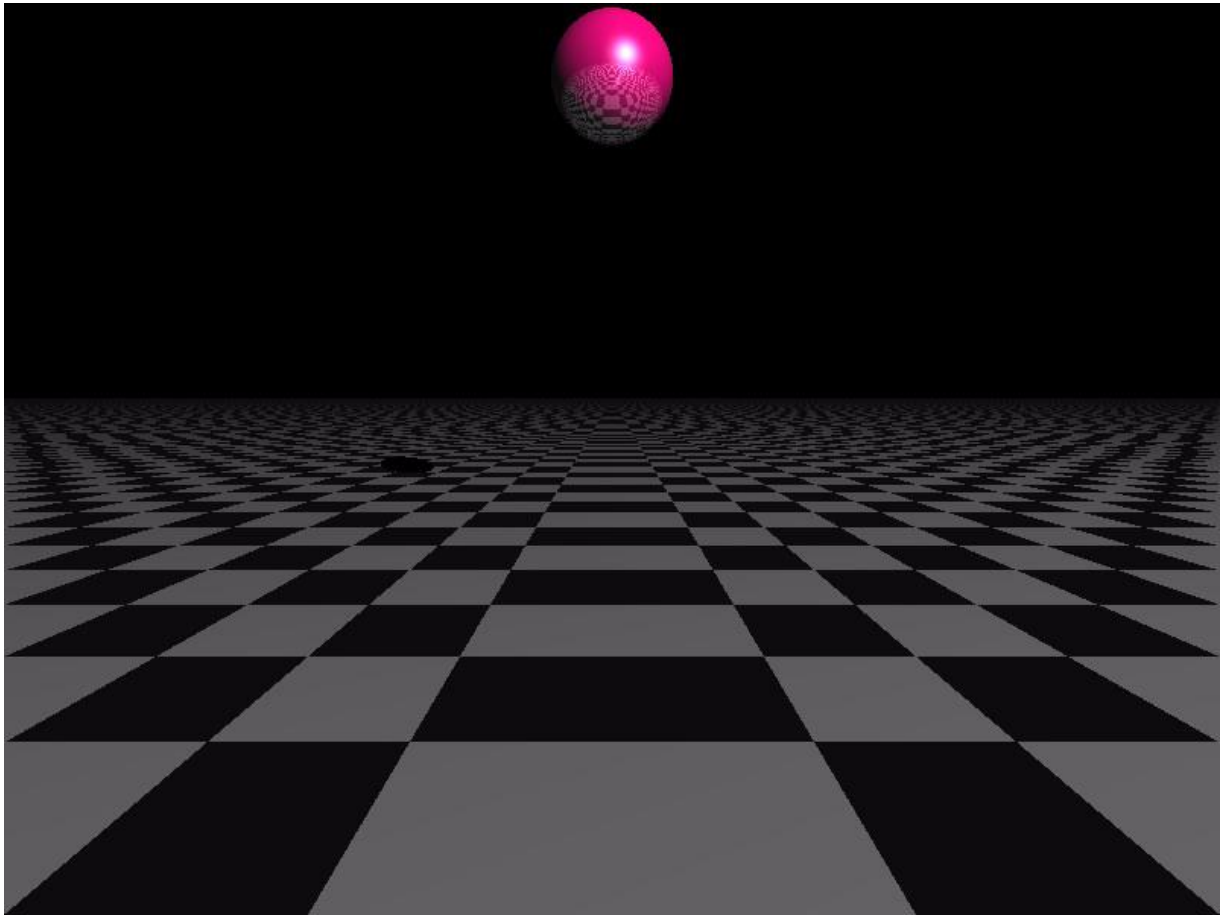
Very simple ray tracing engine in (almost) pure Python. Depends on NumPy and Matplotlib. Diffuse and specular lighting, simple shadows, reflections, no refraction. Purely sequential algorithm, slow execution.

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

w = 800
h = 600

def normalize(x):
    x /= np.linalg.norm(x)
    return x
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

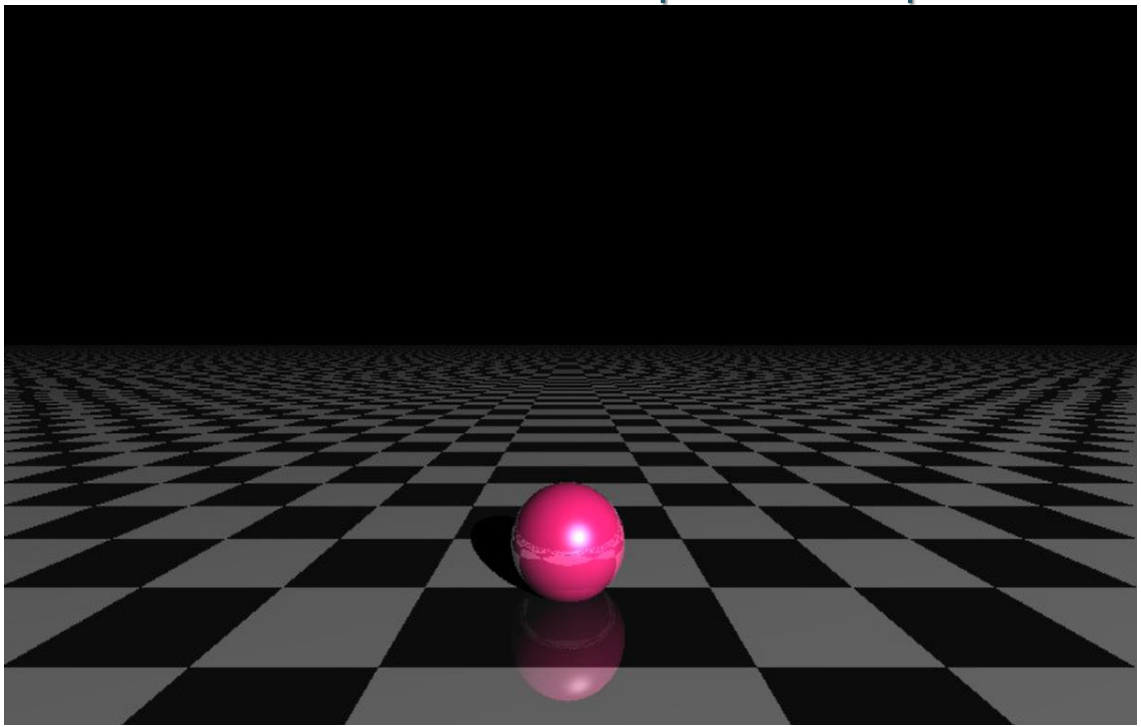


PHYSICS in COMPUTER ANIMATIONS and GAMES

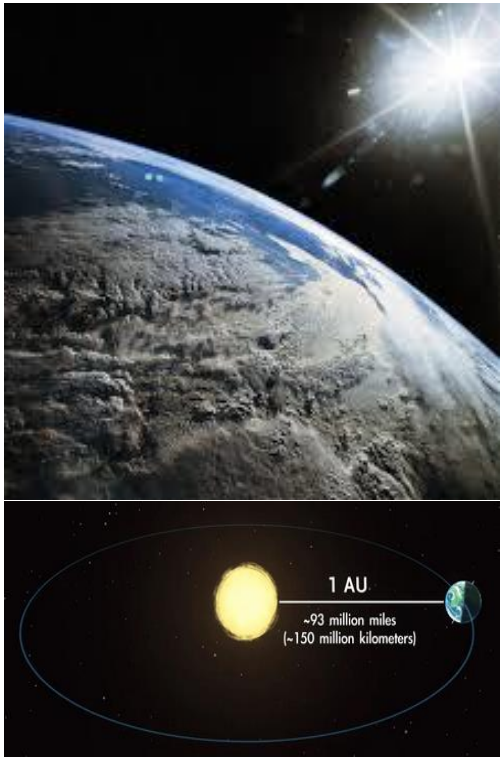


Ray Tracing Homework

Apply the collision calculation between the plane and sphere



2-Body Problem



we want to **simulate the Sun–Earth** system. The semi major axis of the Earth is a =1 AU. For simplicity, let's just assume that the Earth orbits the Sun on a circular orbit, and so the distance between the Earth and the Sun is a constant r = 1 AU. The mass of the Sun and that of the Earth are M and m, respectively. Let G be the universal gravitational constant and r be the position vector of the Earth relative to the Sun. So the gravitational acceleration between the Sun and the Earth is

$$a = -G\frac{M\boldsymbol{r}}{r^3}$$

where the minus sign in the beginning of the equation indicates that the gravitational acceleration is pointing towards the opposite direction of r (i.e., the origin).

The centrifugal force per unit mass of the Earth is

$$a_c = \frac{v^2}{r}$$

where *v* is the linear velocity of the Earth. In order to have the Earth moving on a circular orbit, $|a| = a_c$ must hold everywhere along the orbit.

Equating for gravitational acceleration and centrifugal force $v = \sqrt{\frac{GM}{r}}$ we have

This is often called ***the circular velocity***.

PHYSICS in COMPUTER ANIMATIONS and GAMES

we have the initial conditions ready. If M=1,then m=3 10 6 (the mass ratio between the Sun and the Earth is 3.10^{-6}). Since it is just a toy model, let's assume that G = 1, in which case we have v = 1. It is convenient to assume that the Sun is at the origin of a Cartesian coordinate system with position vector X=[0, 0, 0] and zero velocity, V=[0, 0, 0], then we have x=[1, 0, 0], v=[0, 1, 0].

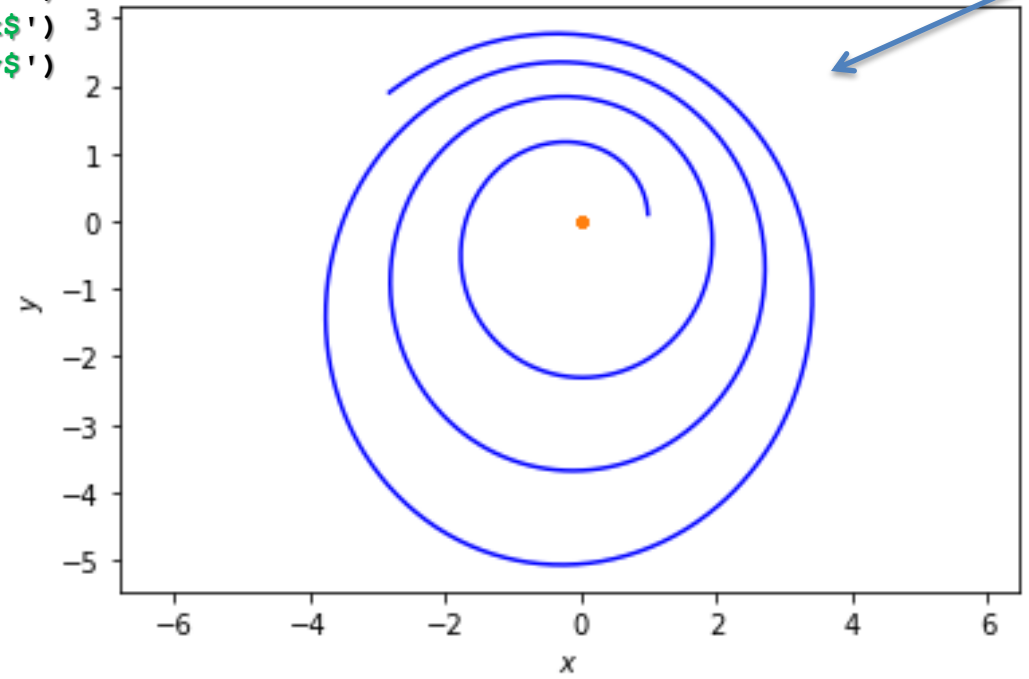
```
import numpy as np
import matplotlib.pyplot as plt

M = 1.0
m = 3.0e-6
G = 1.0

xS = np.array([0., 0., 0.])       # the position vector of the Sun
xE = np.array([1., 0., 0.])       # the position vector of the Earth
vS = np.array([0., 0., 0.])       # the velocity vector of the Sun
vE = np.array([0., 1., 0.])       # the velocity vector of the Earth
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

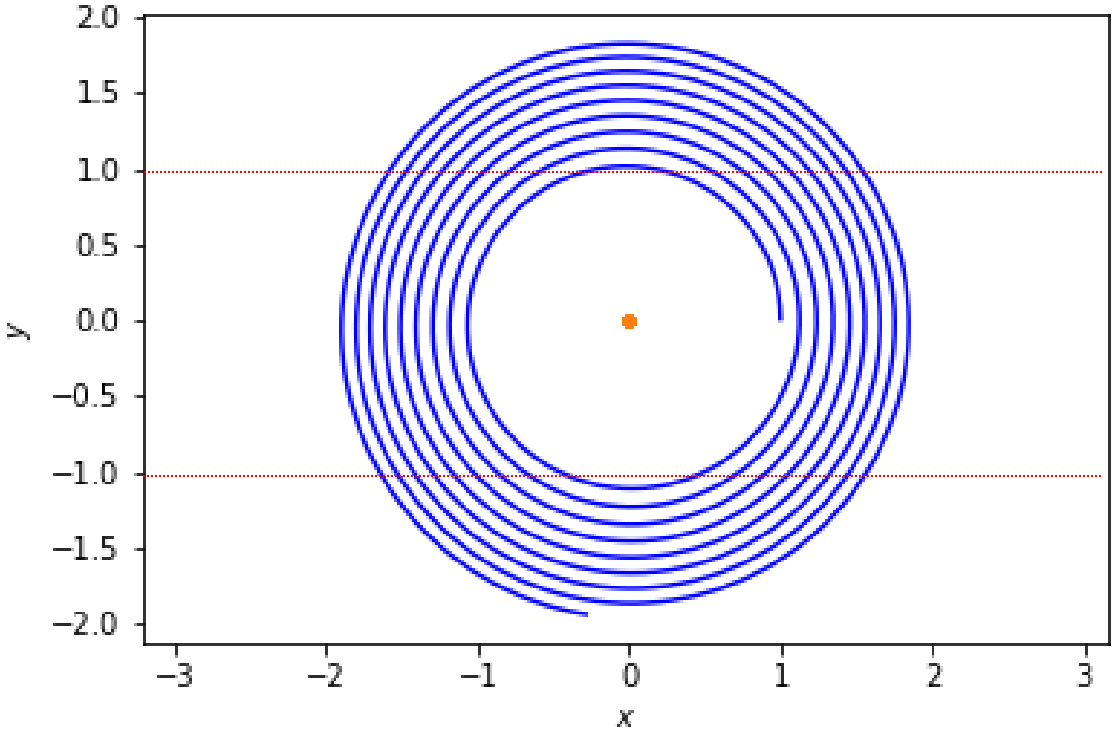
```
dt = 0.1
t_end = 100

xE_vec = [] # x of the Earth
yE_vec = [] # y of the Earth
xS_vec = [] # x of the Sun
yS_vec = [] # y of the Sun

for t in np.linspace(0, t_end, int(t_end/dt)):
    # acc Sun--> Earth
    aE =-G * M * (xE- xS) / np.linalg.norm(xE - xS) ** 3
    # acc Earth--> Sun
    aS =-G * m * (xS- xE) / np.linalg.norm(xS- xE) ** 3
    # advance the positions and velocities
    xE = xE + vE * dt
    xS = xS + vS * dt
    vE = vE + aE * dt
    vS = vS + aS * dt
    # store the data for plotting
    xE_vec.append(xE[0])
    yE_vec.append(xE[1])
    xS_vec.append(xS[0])
    yS_vec.append(xS[1])
```

```
# make the plot
fig, axs = plt.subplots()
plt.plot(xE_vec, yE_vec, 'b')
plt.plot(xS_vec, yS_vec, '.',color='tab:orange')
axs.axis('equal')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```

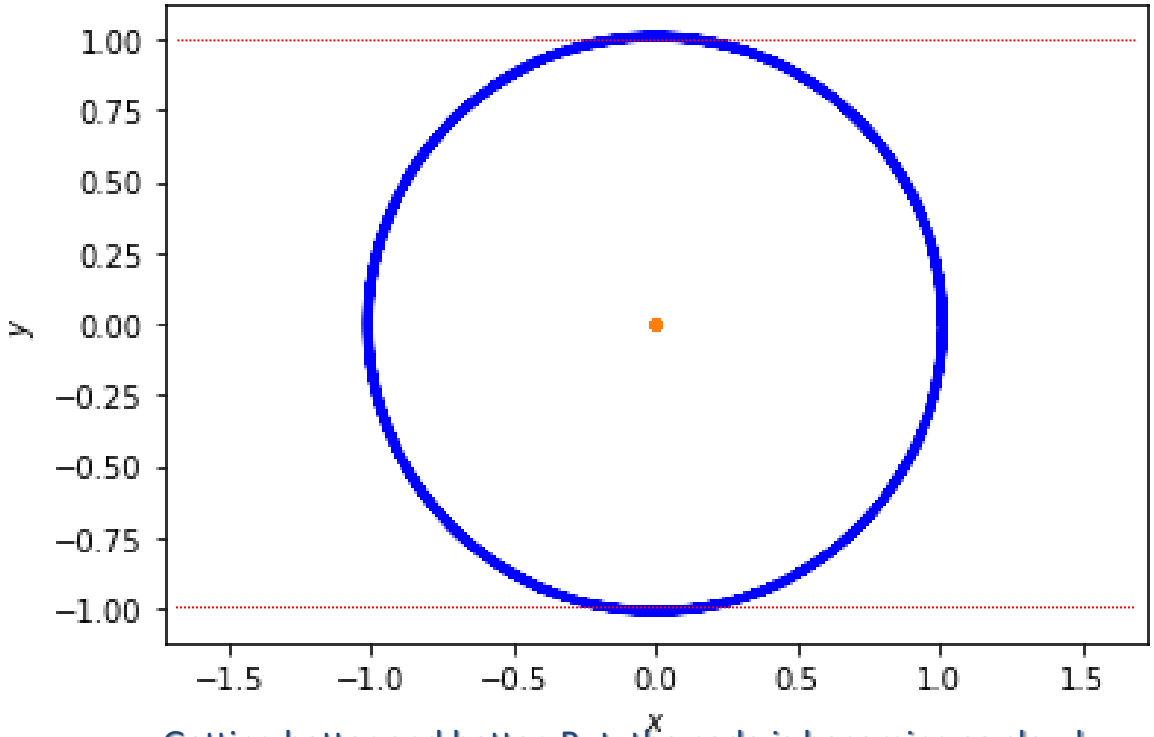
Looks like the Earth is escaping from the Sun!



Decrease the time step by a factor of 10. Now with **dt = 0.01**

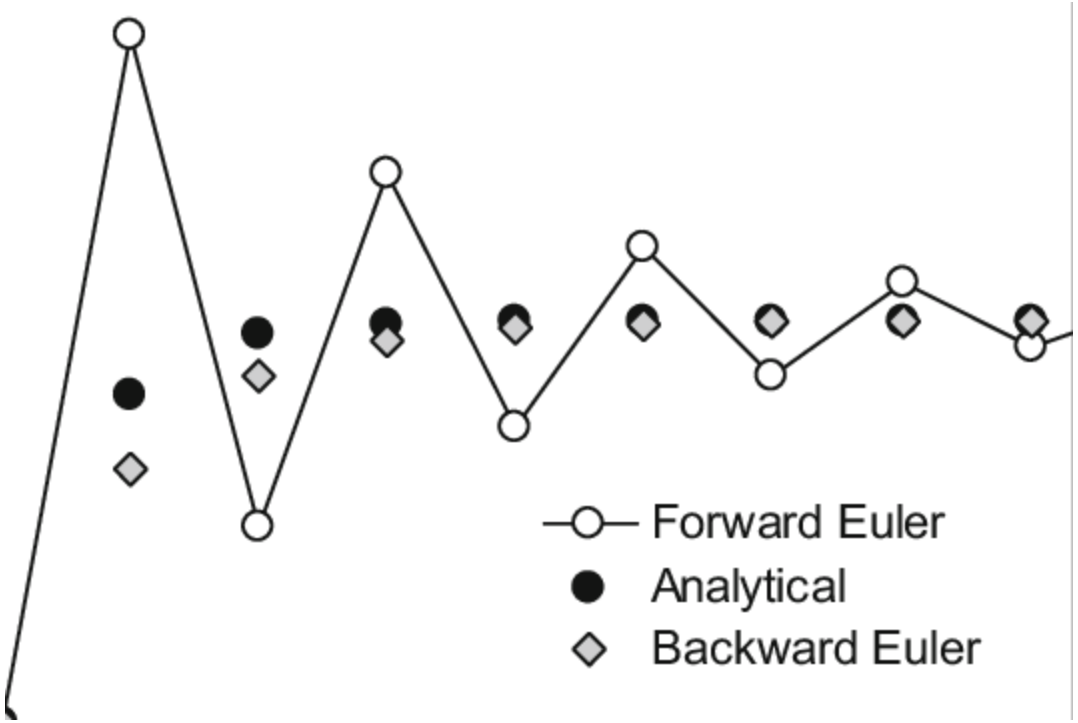


Decrease the time step by a factor of 100. Now with **dt = 0.0001**



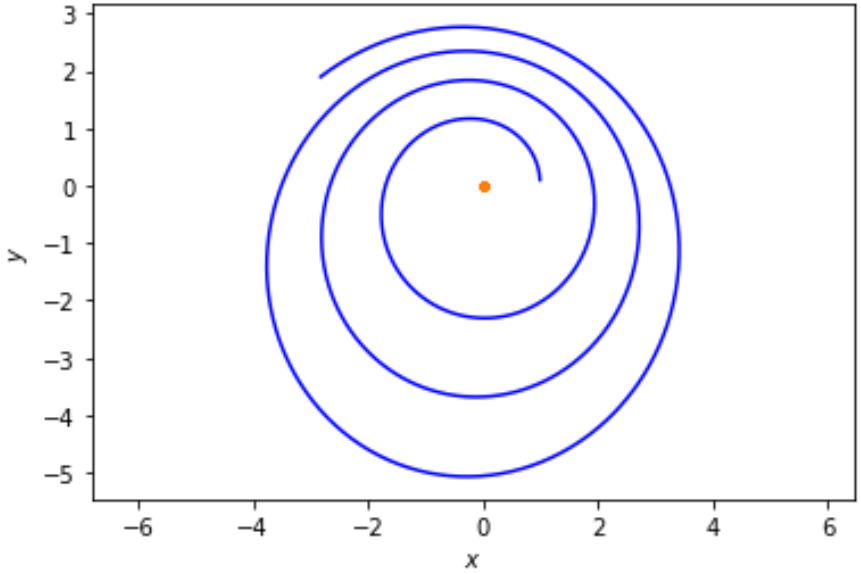
Getting better and better. But, the code is becoming so slow!

```
# Forward Euler
    xE = xE + vE * dt
    xS = xS + vS * dt
    vE = vE + aE * dt
    vS = vS + aS * dt
```



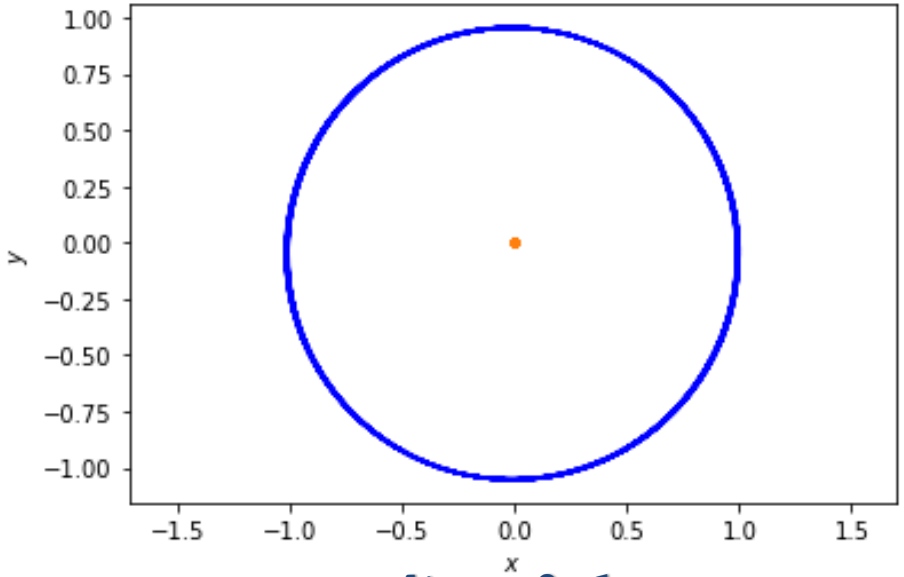
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Forward Euler
    xE = xE + vE * dt
    xS = xS + vS * dt
    vE = vE + aE * dt
    vS = vS + aS * dt
```



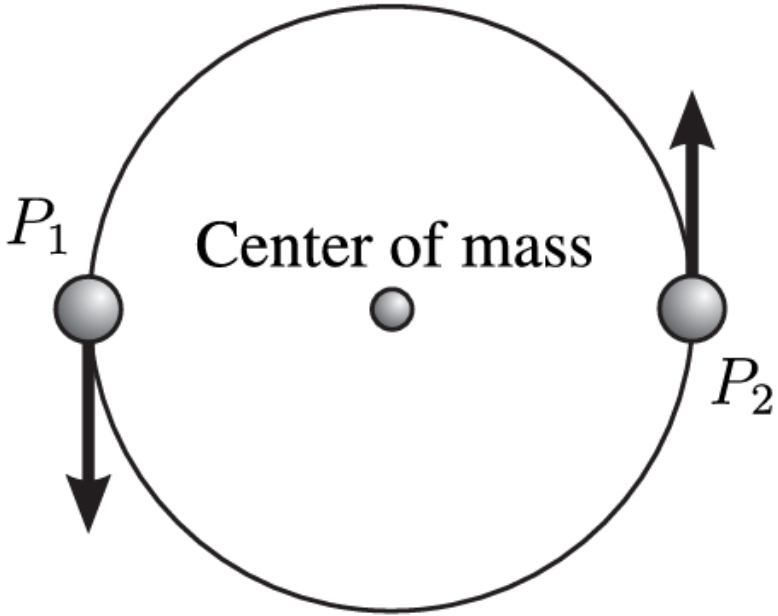
dt = 0.1

```
# Backward Euler
    vE = vE + aE * dt
    vS = vS + aS * dt
    xE = xE + vE * dt
    xS = xS + vS * dt
```

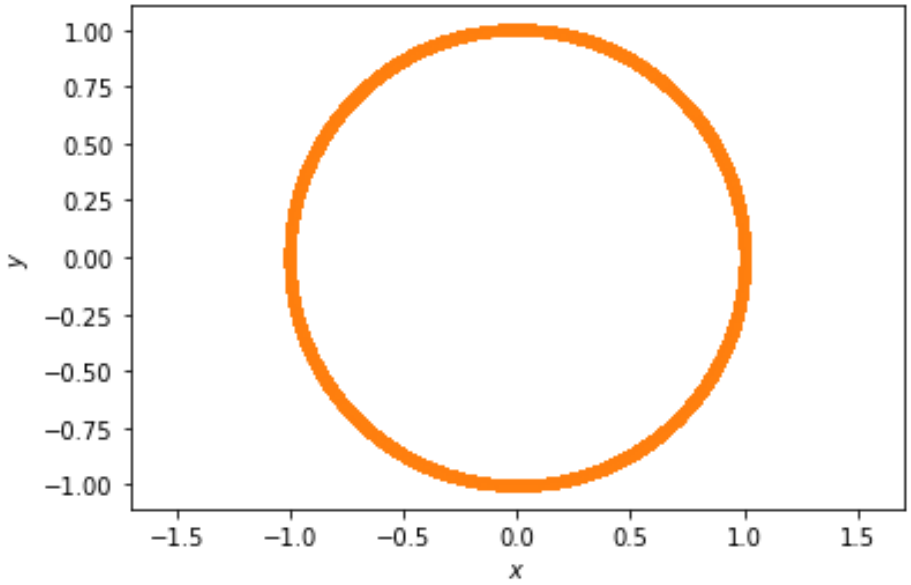


dt = 0.1

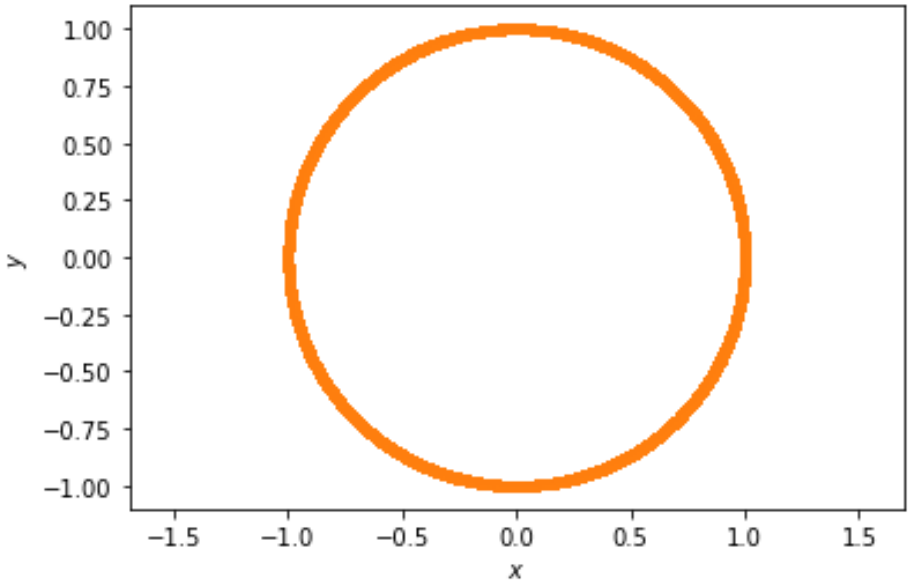
In this configuration, the Sun is much heavier than the Earth. Out of curiosity, let's try another system. As shown in the figure below, let's assume a binary system with two equal-mass particles.




```
M = 1.0          # P1
m = 1.0          # P2

xS = np.array([-1, 0, 0])  # position vector of P1
xE = np.array([1, 0, 0])   # position vector of P2
vS = np.array([0, -0.5, 0])# velocity vector of P1
vE = np.array([0, 0.5, 0]) # velocity vector of P2
```

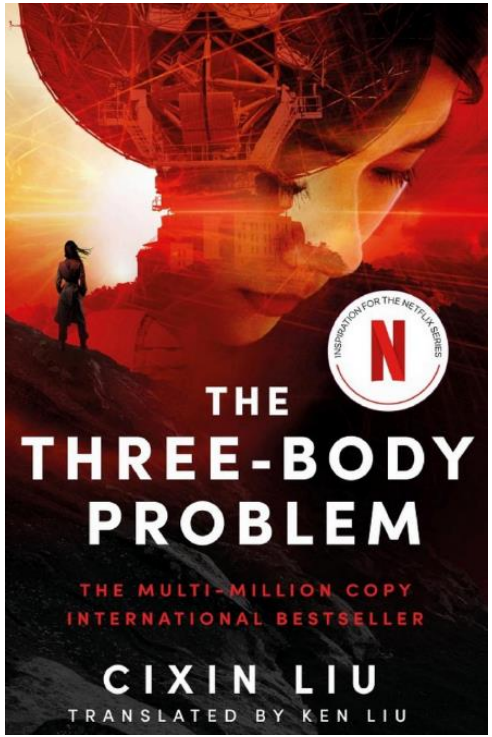


dt = 0.0001



dt = 0.00001

THE THREE-BODY PROBLEM



In classical mechanics, the three-body problem is the problem of taking the initial positions and velocities (or momenta) of three point masses and solving for their subsequent motion according to Newton's laws of motion and Newton's law of universal gravitation.