

# FROM CODE TO COLLISION $+ \cdot$ $\circ$

HOW WE CREATED A 2D PHYSICS  
ENGINE FROM SCRATCH

A Science Society Presentation by Ethan Barnard  
& Andy Zeng



# A Note Before We Begin

---

This talk contains several mathematical explanations aimed at students studying A Level Mathematics and Physics.

---

If you're not taking either of these subjects, some concepts or notation may be challenging - but we've done our best to simplify things so they're understandable to any Year 11-13 STEM student.

# Contents

Introduction – What is a physics engine? Overview of our project.

Demonstration – Simulation of elastic collisions.

Collision Detection – Identifying when and where particles collide.

Collision Resolution – Calculating post-collision velocities.

Fundamentals of Motion & Elastic Collisions – How particles move and interact.

Optimizations & Future Improvements – Enhancing efficiency and scalability.

Real-World Applications – Where similar physics models are used.

Q&A – Questions

# Introduction

- A physics engine is a computer program that simulates physical interactions in a virtual environment.
- This presentation shows a model of particles in a closed system where only elastic collisions occur, conserving momentum and kinetic energy upon collisions.
- We will cover how our engine works, including collision detection, collision resolution, and key challenges.
- By the end, you'll understand how we simulated realistic particle interactions through code.



# Video Demonstration of the Engine Working



# Mathematics of Motion

**In a closed system, like our 2D particle simulator, motion follows three key principles:**

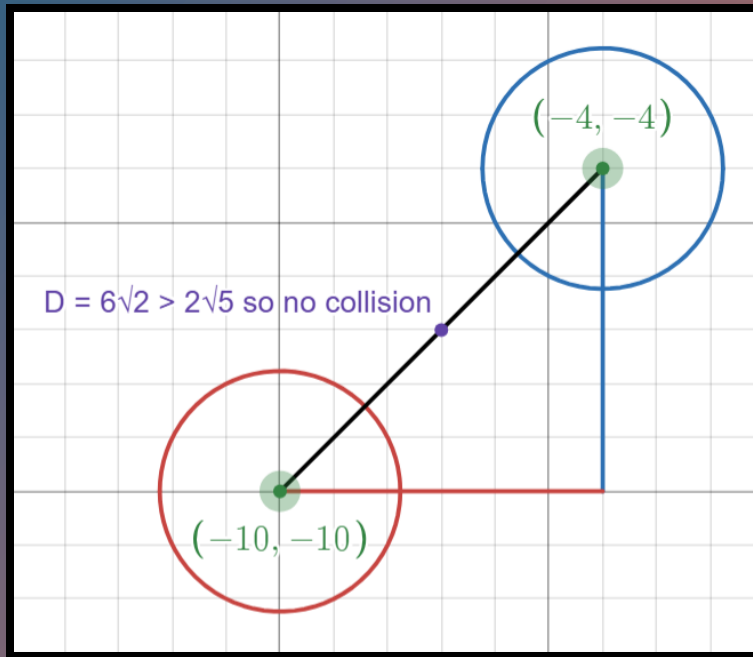
1. Particles move freely with constant velocity, as no external forces (e.g., gravity or friction) are applied.
2. Each particle has three fundamental quantities: **position**, **velocity** and **mass**.
3. Motion is updated using **position, velocity and time** via this equation:

Given the time step index  $N$ , the position vector  $r$ , velocity vector  $v$  and the time  $t$ :

$$\vec{r}_N = \vec{r}_{N-1} + \vec{v}_{N-1} * \delta t$$

This states that the new position equals the old position plus the distance travelled between positions





# Mathematics of Collision Detection

In this particle simulator, as particles are modelled as circles, a simple rule can be used to check if a collision has occurred.

- $distance = \sqrt{\delta x^2 + \delta y^2}$
- If  $radius_{obj1} + radius_{obj2} \leq distance$ , then no collision has occurred, and we can ignore any collision resolution. This is to reduce the number of unnecessary force calculations.

If there is an overlap between the objects, then we must resolve the collision.

```
def check_collision(obj1, obj2):  
    """Checks and handles collision between two objects."""  
    dx, dy = obj2.x - obj1.x, obj2.y - obj1.y  
    distance = (dx**2 + dy**2) ** 0.5  
    min_distance = obj1.radius + obj2.radius  
  
    if distance >= min_distance:  
        return
```

# Mathematics of Collision Resolution

To resolve the collision, a normal vector  $\hat{n}$  is defined from the centre of one particle to another. The circumflex symbol means that this is a unit vector, so it has a magnitude/ length of 1 unit. Let's look at how object 1 (of the two colliding objects) is handled:

The tangent unit vector  $\hat{t}$  is perpendicular to  $\hat{n}$ .

$\vec{v}_1$  is the velocity of object 1.

$v_{1n}$  is the normal velocity in the collision.

- $v_{1n} = \vec{v}_1 \cdot \hat{n}$  for the velocity component along the normal. This is what changes in the collision.
- $v_{1t} = \vec{v}_1 \cdot \hat{t}$  for the velocity component along the tangent. This is what stays constant.



# Diagrammatic Representation

$v_{1n}$  controls how fast object 1 is heading towards or away from object 2.

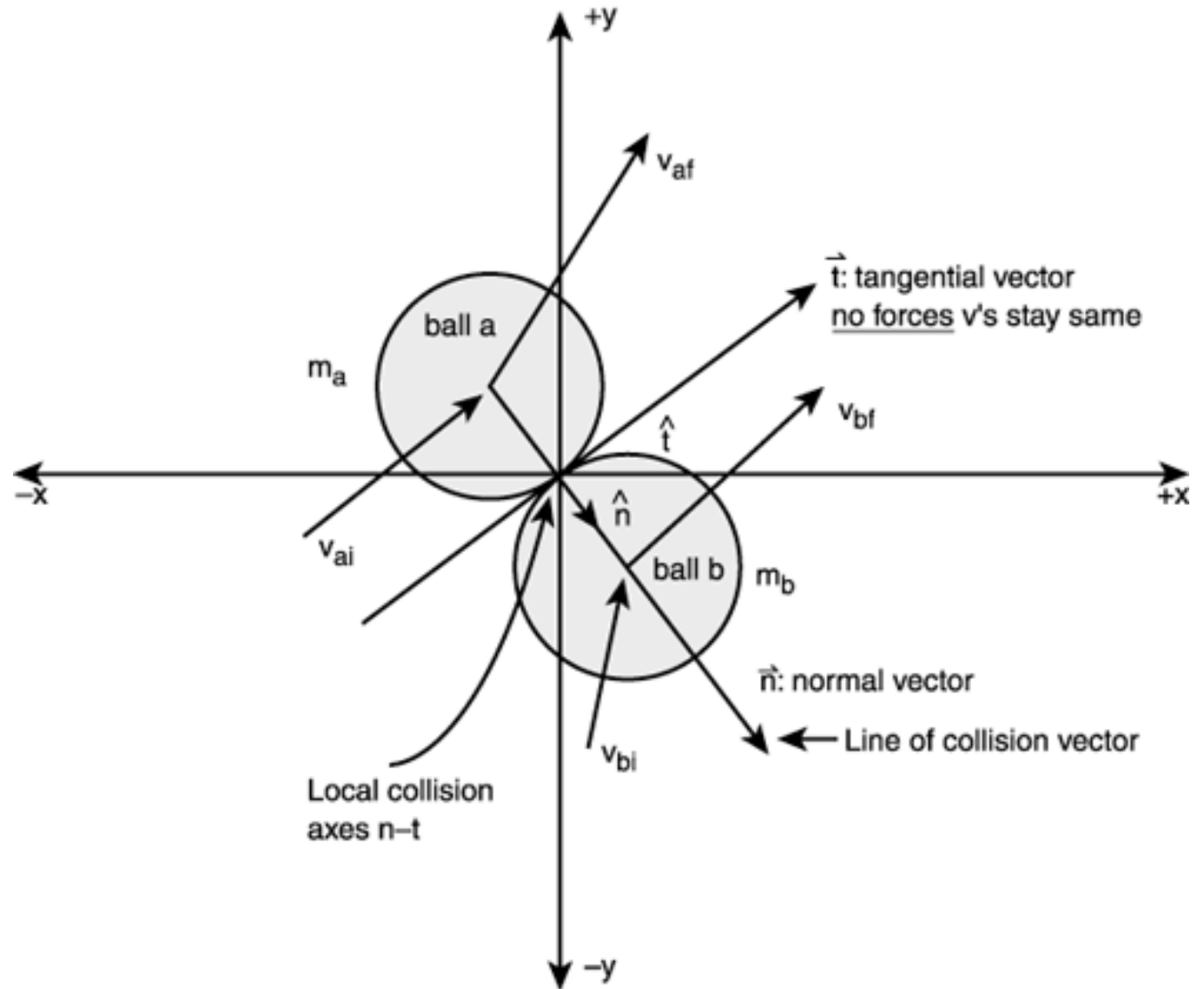
$v_{1t}$  controls how fast object 1 is moving along the surface of object 2. As these are modelled as smooth and frictionless, this stays constant.

- $v'_{1n}$  is the normal component of object 1's velocity after collision

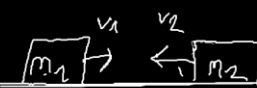
- $$v'_{1n} = \frac{(m_1 - m_2) v_{1n} + 2m_2 v_{2n}}{m_1 + m_2}$$

The final velocity vector of object 1 after the collision is as follows:

- $$\vec{v}'_1 = v'_{1n} \hat{n} + v_{1t} \hat{t}$$



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



$$m_1 v_1 + m_2 v_2 = m_1 v'_1 + m_2 v'_2 \rightarrow m_1 (v_1 - v'_1) = m_2 (v'_2 - v_2)$$

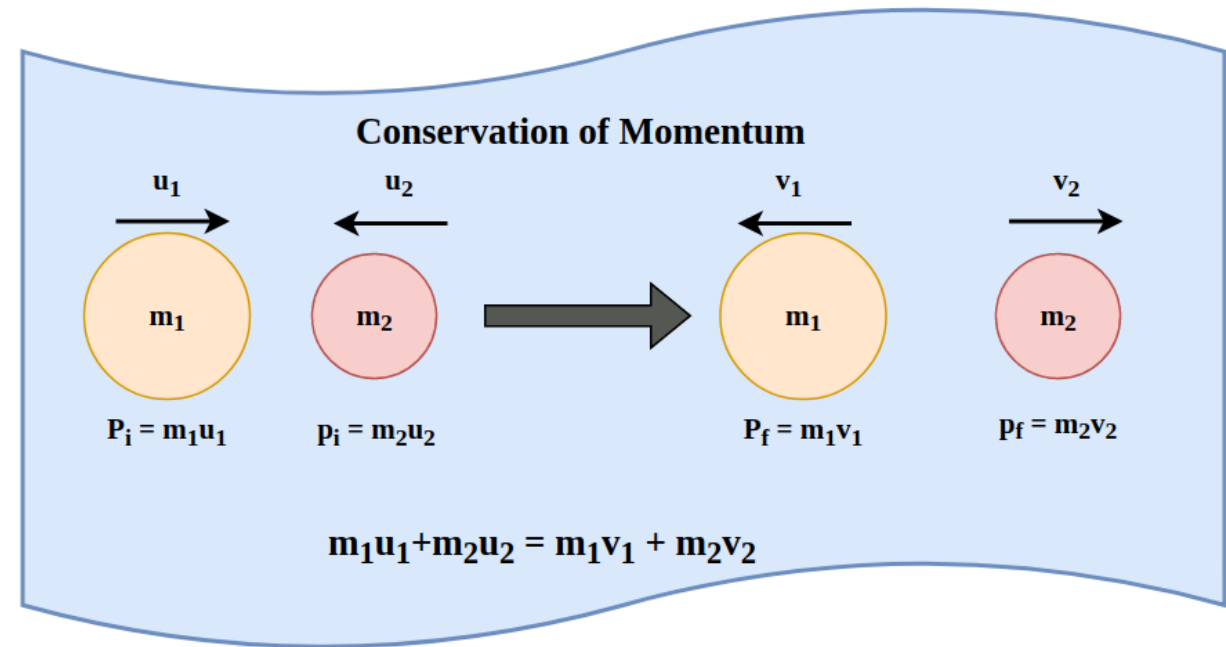
$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 v'^2_1 + \frac{1}{2} m_2 v'^2_2 \rightarrow m_1 (v_1^2 - v'^2_1) = m_2 (v'^2_2 - v_2^2)$$

$$m_1 (v_1 - v'_1)(v_1 + v'_1) = m_2 (v'_2 - v_2)(v'_2 + v_2) \rightarrow v_1 + v'_1 = v'_2 + v_2$$

$$v'_2 = v_1 - v'_1 + v_2$$

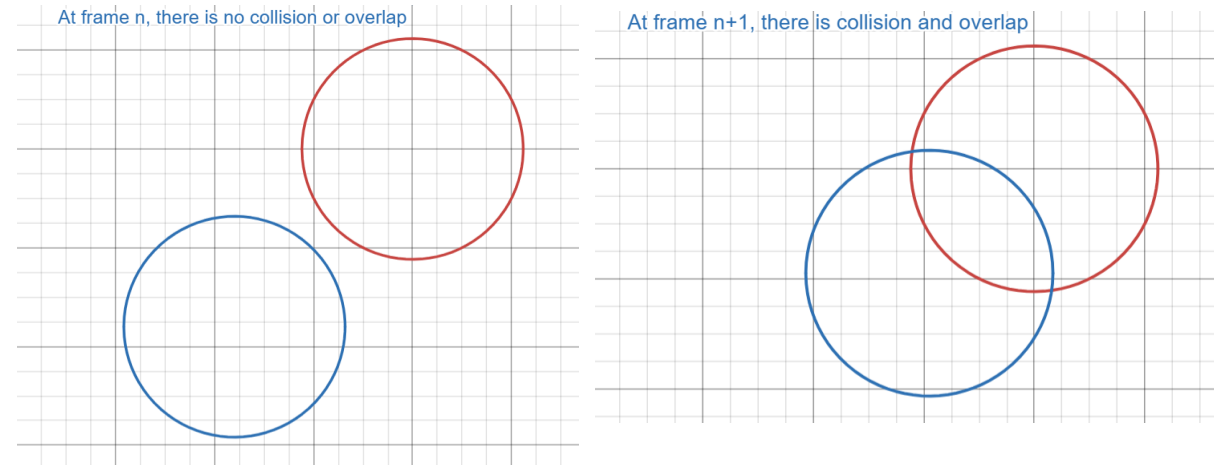
$$m_1 v_1 + m_2 v_2 = m_1 v'_1 + m_2 (v_1 - v'_1 + v_2)$$

$$v'_1 = \frac{2m_2 v_2 + v_1 (m_1 - m_2)}{m_1 + m_2}$$



# Collision Resolution Continued/ Recap

- A normal vector is created, representing the direction of impact between the two objects colliding.
- The new velocities after collision are calculated from the normal component of the velocities.
- The objects could overlap each other upon collision, rather than just touching, so this must be taken into consideration to prevent this from negatively affecting the collision algorithm. This is a limitation in simulating physics, as the program can only check for collisions at every frame, and not between frames.



```
normal = [dx / distance, dy / distance] if distance != 0 else [1, 0]
tangent = [-normal[1], normal[0]]

v1n, v1t = obj1.x_vel * normal[0] + obj1.y_vel * normal[1],
obj1.x_vel * tangent[0] + obj1.y_vel * tangent[1]
v2n, v2t = obj2.x_vel * normal[0] + obj2.y_vel * normal[1],
obj2.x_vel * tangent[0] + obj2.y_vel * tangent[1]

v1n_after = (v1n * (obj1.mass - obj2.mass) + 2 * obj2.mass * v2n) /
(obj1.mass + obj2.mass)
v2n_after = (v2n * (obj2.mass - obj1.mass) + 2 * obj1.mass * v1n) /
(obj1.mass + obj2.mass)

obj1.x_vel = v1n_after * normal[0] + v1t * tangent[0]
obj1.y_vel = v1n_after * normal[1] + v1t * tangent[1]
obj2.x_vel = v2n_after * normal[0] + v2t * tangent[0]
obj2.y_vel = v2n_after * normal[1] + v2t * tangent[1]

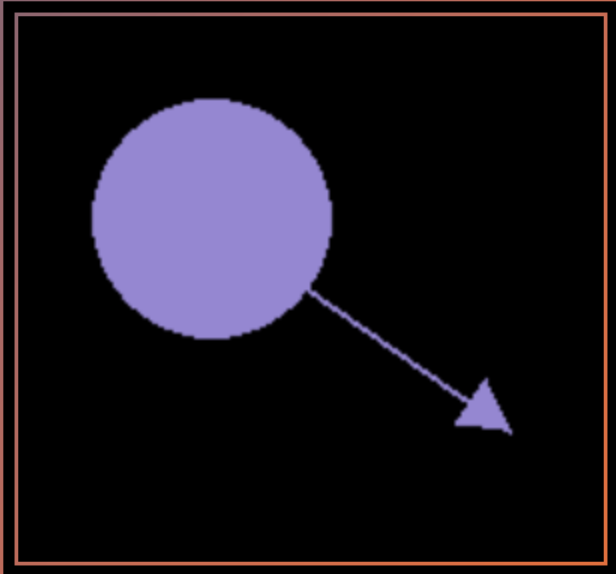
overlap = min_distance - distance
obj1.x -= overlap * normal[0] / 2
obj1.y -= overlap * normal[1] / 2
obj2.x += overlap * normal[0] / 2
obj2.y += overlap * normal[1] / 2
```

# Launching Vector

```
def calculate_vector(mouse_trajectory, scaling_factor=20,
min_length=10):
    """Calculates the vector for mouse movement based on trajectory."""
    if len(mouse_trajectory) < min_length:
        return 0, 0

    start_x, start_y = mouse_trajectory[0][0], mouse_trajectory[1][1]
    end_x, end_y = mouse_trajectory[-1][0], mouse_trajectory[-1][1]

    return (end_x - start_x) / scaling_factor, (end_y - start_y) /
scaling_factor
```



- When launching objects onto the screen, all previous positions of the mouse can be tracked by an algorithm.
- This can be used to determine the path of your mouse as you create a trajectory for the object.
- A scaling factor is used so that the length of the trajectory is proportional to the velocity of the object
- We'll need this later for the particle movement slide.

# Particle Class

```
class Particle:
    def __init__(
        self, x: int, y: int, x_vel: float, y_vel: float, mass: float,
        color: Tuple[int, int, int], radius: float
    ) -> None:
        self.x: int = x
        self.y: int = y
        self.x_vel: float = x_vel
        self.y_vel: float = y_vel
        self.mass: float = mass
        self.color: Tuple[int, int, int] = color
        self.radius: float = radius
        self.selected: bool = False
        self.shape: draw.Rect | None = None
```

- This acts as a template to define the state and behaviour of each particle that is created in the simulation.
- Important physical quantities are declared here, such as coordinates, velocity, mass, colour, radius, whether the user is holding the object or not, and the shape of the particle. In this simulator, all particles are circular.

# Moving a Particle

- Firstly, this block of code checks if the particle has been selected by the user.
- If so, the program must calculate the vector of the object (as analysed on the launching vector slide). Otherwise, the program ensures that the particle rebounds off the side walls when necessary. An example rule for the rebound in the x direction, written in 'pseudocode' is:
- ( $v_x$  is the velocity in the x direction)

$$\begin{array}{c} \text{if} \\ (x - radius_{objn} + v_x < 0) \\ \text{or} \end{array}$$

$$\begin{array}{c} (x + radius_{objn} + v_x > ScreenWidth) \\ \text{then} \end{array}$$

$$v_x = v_x * -1$$

- This looks at where the particle will be on the next frame and checks if it will be out of bounds. If so, the velocity can be negated to ensure the object stays in bounds in the next frame.

```
def move(self, pos: Tuple[int, int], mouse_down: bool) -> None:
    if self.shape and self.shape.collidepoint(pos) and mouse_down:
        self.selected = True
    elif not mouse_down:
        self.selected = False

    def update_position(self, pos: Tuple[int, int], mouse_trajectory:
    Tuple[float, float], particles_list: List["Particle"]) -> None:
        if self.selected:
            res = calculate_vector(mouse_trajectory)
            self.x_vel = res[0]
            self.y_vel = res[1]
            self.x = pos[0]
            self.y = pos[1]

            for event_instance in event.get():
                if (event_instance.type == KEYDOWN and
                    event_instance.key == K_BACKSPACE) or
                    (event_instance.type == KEYDOWN and
                     event_instance.key == K_DELETE):
                    if self in particles_list:
                        print(f"[I] Deleted particle at [{self.x},
                        {self.y}] with x velocity {str(self.x_vel)} ms-1, y velocity
                        {str(self.y_vel)} ms-1 and mass {self.mass} kg!")
                        particles_list.remove(self)
                    else:
                        # Check boundaries for x-axis
                        if self.x - self.radius + self.x_vel < 0 or self.x +
                        self.radius + self.x_vel > WIDTH:
                            self.x_vel *= -1

                        # Check boundaries for y-axis
                        if self.y - self.radius + self.y_vel < 0 or self.y +
                        self.radius + self.y_vel > HEIGHT:
                            self.y_vel *= -1

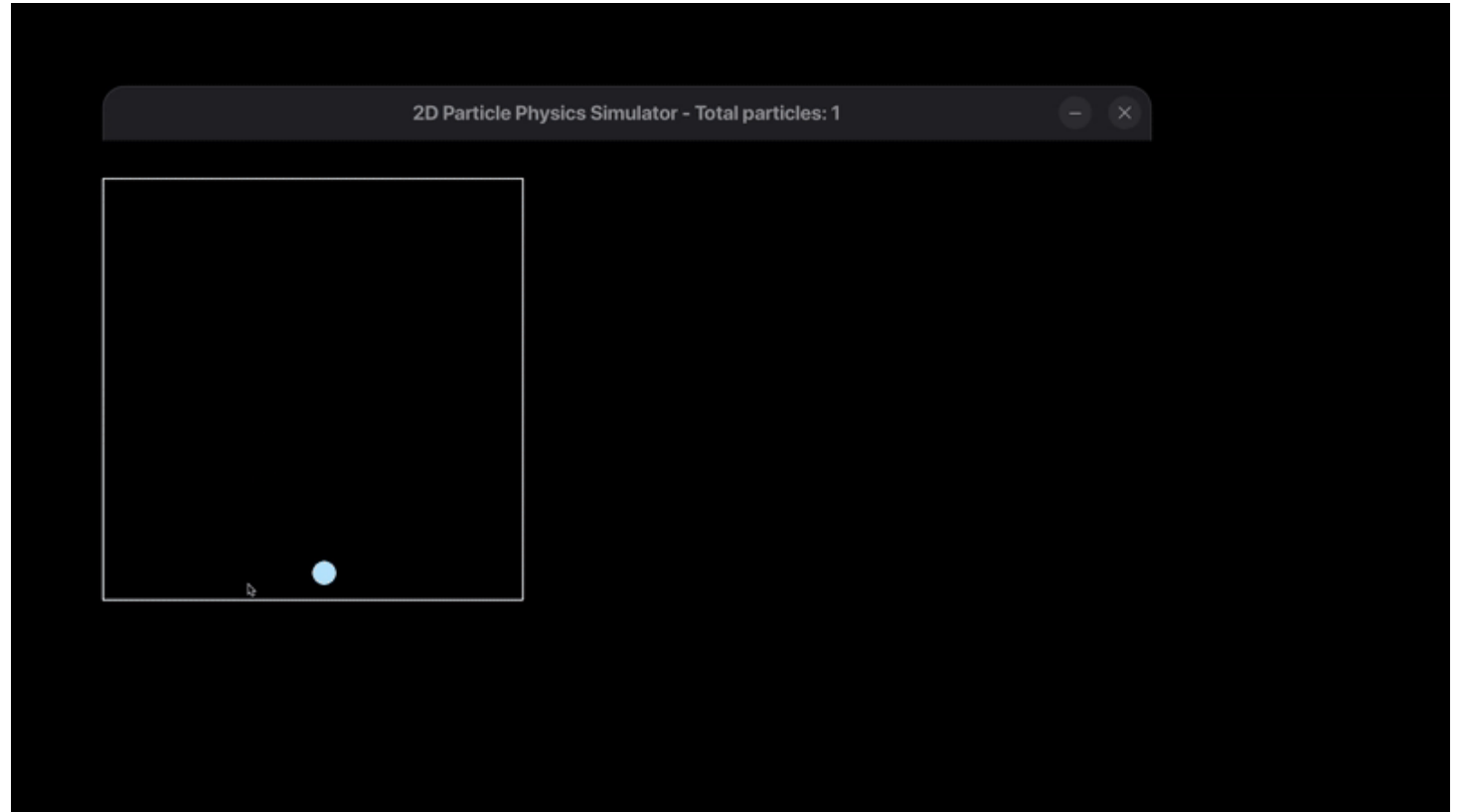
            self.x += self.x_vel
            self.y += self.y_vel
```

# Visualisation of boundary collision

---

If the object is moving out the left boundary in the next frame, then the object needs to change direction to stay in bounds.

This is shown visually on the right.



# Particle Analysis

```
def draw_attributes(self, screen: Surface) -> None:
    attributes = [
        f"XVel: {self.x_vel:.2f} ms-1",
        f"YVel: {self.y_vel:.2f} ms-1",
        f"Momentum: {self.get_current_momentum():.2f} kgms-1",
        f"Mass: {self.mass:.2f} kg",
    ]

    for i, attribute in enumerate(attributes):
        text = base_font.render(attribute, True, (255, 255, 255),
(0, 0, 0)) # White text with black background
        text_rect = text.get_rect(center=(self.x, self.y -
self.radius - 20 + i * 20))
        screen.blit(text, text_rect)

def draw_arrows(self, screen: Surface) -> None:
    end_x = int(self.x + self.x_vel * arrow_scale)
    end_y = int(self.y + self.y_vel * arrow_scale)

    draw_arrow(screen, self.color, (int(self.x), int(self.y)),
(end_x, end_y), arrow_size=arrow_scale, radius=self.radius)

def draw(self, screen: Surface) -> None:
    self.shape = draw.circle(screen, self.color, (int(self.x),
int(self.y)), int(self.radius))
```

- The current analysis of each object is a toggleable feature for the user.
- The attributes shown for each object include its velocity components, momentum and mass.
- To analyse every particle, a loop must be used to search through each object on the screen and to display the properties. This will have white text with a black background.
- To indicate the particle direction, an arrow will be drawn on the screen connected to the particle. This part of the code is abstracted, as further details on arrow creation will be given on the next slide.



# Walls and Arrows

```
def draw_walls(screen, width, height):
    """Draws the boundary walls of the simulator"""

    color = 'white'
    thickness = 5
    draw.line(screen, color, (0, 0), (width, 0), thickness) # Upper
    wall
    draw.line(screen, color, (0, height), (width, height), thickness)
    # Lower wall
    draw.line(screen, color, (0, 0), (0, height), thickness) # Left
    wall
    draw.line(screen, color, (width, 0), (width, height), thickness) #
    Right wall

def draw_arrow(surface: Surface, color: str, start: Tuple[float,
float], end: Tuple[float, float],
               arrow_size: int, radius: int, thickness=2):
    """Draws an arrow from start to end, ensuring it does not phase
    into a particle."""

    limit = radius + 1 # Adjusted limit
    start_vec = Vector2(start)
    end_vec = Vector2(end)

    # Compute the direction vector
    direction = end_vec - start_vec
    distance = direction.length()

    # Prevent division by zero and avoid drawing when the particle is
    selected
    if distance == 0:
        return

    # Ensure the arrow stops before hitting the particle
    if distance > radius:
        new_end_vec = start_vec + direction.normalize() * (distance +
radius)
    else:
        return

    # Draw the main arrow line only if it's long enough
    if distance > limit:
        draw.line(surface, color, start, (new_end_vec.x,
new_end_vec.y), thickness)

    # Compute the angle of the arrow
    angle = atan2(direction.y, direction.x)

    # Position the arrowhead at `new_end_vector`
    arrow_point1 = (new_end_vec.x - arrow_size * cos(angle - pi / 6),
new_end_vec.y - arrow_size * sin(angle - pi / 6))
    arrow_point2 = (new_end_vec.x - arrow_size * cos(angle + pi / 6),
new_end_vec.y - arrow_size * sin(angle + pi / 6))

    # Always draw the arrowhead
    draw.polygon(surface, color, [(new_end_vec.x, new_end_vec.y),
arrow_point1, arrow_point2])
```

- The walls are defined as boundaries for the simulation. This is to prevent any objects from going off the screen.
- The arrow creation is fully defined here. If an arrow is too short, the arrow doesn't need to be created. Otherwise, an arrow is drawn with a size proportional to the magnitude of its velocity.
- There is no built-in arrow creation functionality, so the line of the arrow and the arrow-head is mathematically calculated and created dynamically.

# Camera Functionality

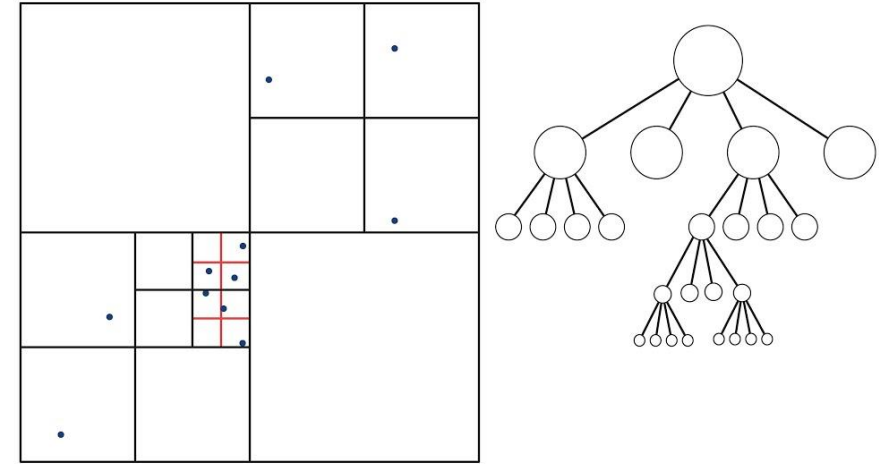
To develop the camera zooming and panning, the real coordinates held by each particle need to be mapped to camera coordinates. These can be used as the inputs to any drawing functionality.

- $x_{\text{zoomed}} = \text{int}((x - \text{camera}_{x \text{ offset}}) * \text{camera}_{\text{zoom}})$
- $y_{\text{zoomed}} = \text{int}((y - \text{camera}_{y \text{ offset}}) * \text{camera}_{\text{zoom}})$
- $\text{radius}_{\text{zoomed}} = \text{int}(\text{radius} * \text{camera}_{\text{zoom}})$

# Optimisations and Future Improvements

- Currently, we check every object against every other object is inefficient in terms of processing time. Using a quadtree structure (shown on the right) is much more efficient, as not every object will need to be compared against every other object, but rather just the ones within the same square.
- We also simulate inelastic collisions using a coefficient of restitution. This is a value between 0 and 1 as a ratio of the speed after collision to before the collision. Additionally, friction can be included to affect the tangential vector.
- A 3D physics engine could include both these features and further increase the realism of the program.

Example



## Coefficient of Restitution

$$e = \frac{v_1' - v_2'}{v_2 - v_1}$$

Elastic

$$e = 1$$

Inelastic

$$0 < e < 1$$

# Real-World Applications

- **Teaching:** Helps students visualise Newton's laws and collisions (e.g. Newton's First and Third Laws)
- **Game Development:** Used for dynamic effects such as explosions, smoke, and debris
- **UI & Web Visual Effects:** Commonly used to create interactive particle effects on websites and apps
- **Aesthetics:** Visually pleasing, runs smoothly in the background with minimal memory and CPU usage – may be paired with animated wallpaper

# Source code

- The full source code for the 2D Physics Engine, including detailed explanations of each function and procedure, is available on GitHub at:  
<https://github.com/aazz18/2d-physics-engine>
- The project is implemented in Python and uses the PyGame library for graphics.



---

Q&A: Any questions?