8 Physics Simulations

8.1 Billiard-Game Physics

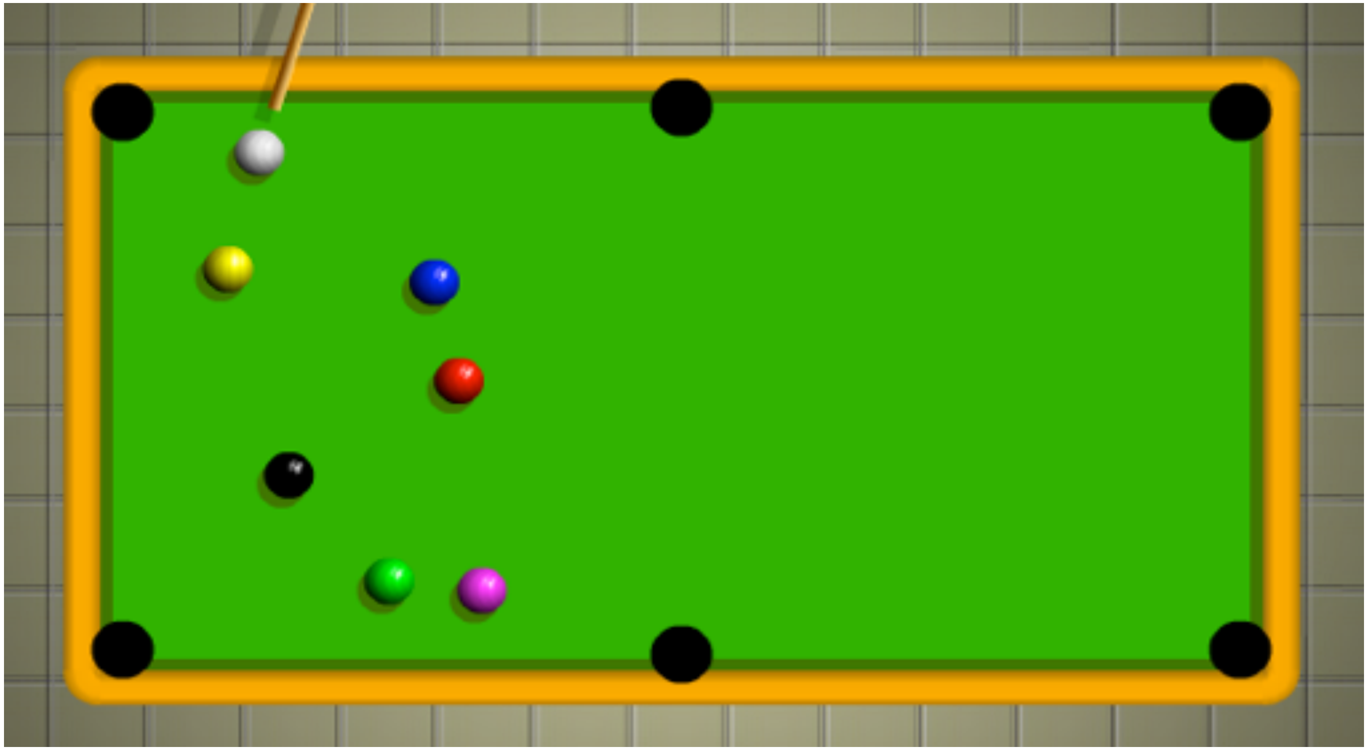
8.2 Game Physics Engines

Literature:

K. Besley et al.: Flash MX 2004 Games Most Wanted,
Apress/Friends of ED 2004 (chapter 3 by Keith Peters)

Billiard-Game Physics

- Typical problem:
  - Two round objects moving at different speeds and angles hitting each other
  - How to determine resulting speeds and directions?
- Classical example: Billiard game

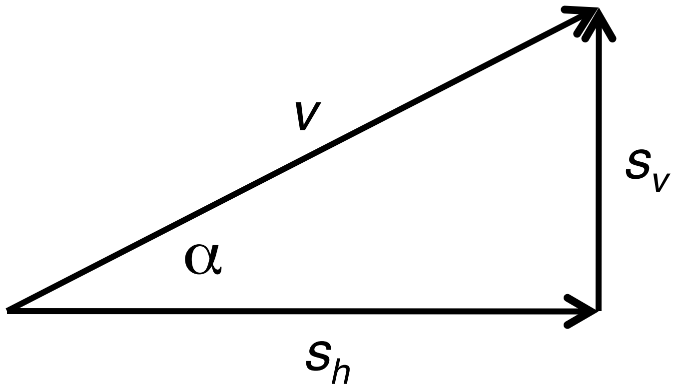


Speed and Velocity

- Speed:
  - Magnitude (single number), measured in px/s
  - Suitable for movement along one axis (e.g. x axis)
- Velocity:
  - Speed plus direction

    Magnitude (px/s) and angle (degrees)
  - Expressed as a 2D vector:

    velocity = (*horizontal_speed*, *vertical_speed*)



$$s_h = \cos(\alpha) \cdot v$$

$$s_v = \sin(\alpha) \cdot v$$

$$v = \sqrt{s_h^2 + s_v^2}$$

$$\alpha = \operatorname{atan}\,(s_v / s_h)$$

Velocity and Acceleration

- Velocity
  - is added to the position values in each frame cycle
- Acceleration
  - is a force *changing velocity*
  - *Acceleration* is added to velocity in each frame cycle
  - *Deceleration* is negative acceleration
- Angular acceleration
  - Acceleration is a 2D vector (or a magnitude plus angle)

```
vx += ax
vy += ay
x += vx
y += vy
```

(*ax*, *ay*) acceleration
(*vx*, *vy*) velocity

Object Orientation for Animated Graphics

- Each moving part of the scene is an *object* (belonging to a class)
- Class definitions comprise:
  - Reference to graphical representation
  - Properties mirroring physical properties (e.g. position, speed)
  - Properties and methods required for core program logic
- Two ways of referring to graphics framework:
  - *Subclassing:* Moving object belongs to a subclass of a framework class
    - e.g. Sprite class, graphics node in scene graph
  - *Delegation:* Moving object contains a reference to a framework object
- Decision criterion:
  - Wanted/needed degree of decoupling between multimedia framework and application logic

Main program for Moving Balls (1)

```
# Create ball 1
b1 = Ball(20,20,ballsize,red)
b1.setBounds(0,sc_w,0,sc_h)
b1.setVelocity(10,15)

# Create ball 2
b2 = Ball(sc_w-20,sc_h-20,ballsize,blue)
b2.setBounds(0,sc_w,0,sc_h)
b2.setVelocity(-20,-10)

# Set frame rate
framerate = 30 # frames per second

clock = pygame.time.Clock()

…(contd.)
```

Ball constructor:
Position, radius, color

Bounds:
“Walls” limiting movement

Main program for Moving Balls (2)

```
… (contd.)

running = True
while running:
      for event in pygame.event.get():
            if event.type == QUIT:
                  running = False
      pygame.draw.rect(screen, white, Rect((0, 0), (sc_w, sc_h)))
      b1.update()
      b1.draw(screen)
      b2.update()
      b2.draw(screen)
      clock.tick(framerate)
      pygame.display.update()
```

Ball Class (Excerpt)

```
class Ball:

    def __init__(self,startX,startY,radius,color):
        self.posX = startX
        self.posY = startY
        self.radius = radius
        self.color = color

    def setVelocity(self,vX,vY):
        self.vX = vX
        self.vY = vY

    def update(self):
        self.posX = self.posX + self.vX
        self.posY = self.posY + self.vY
```

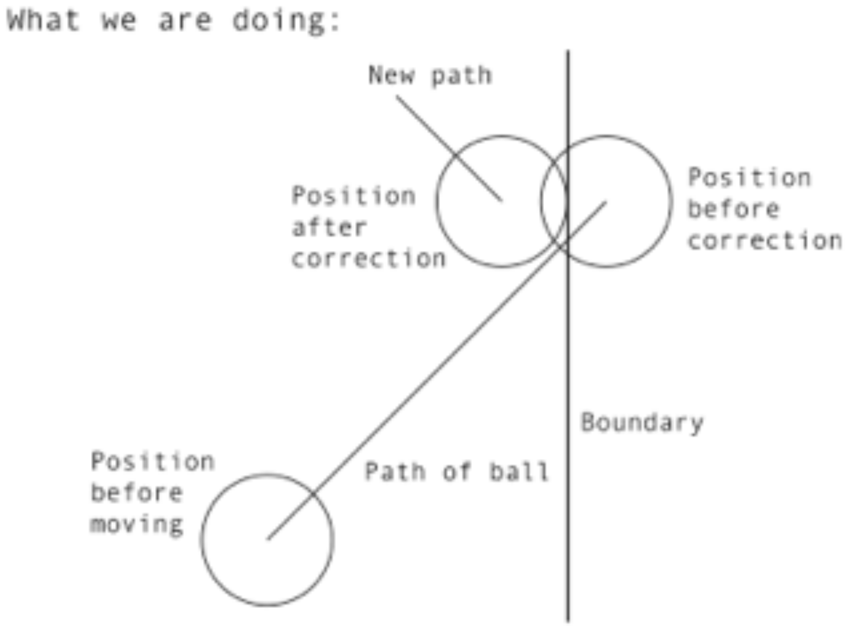
Collision Detection

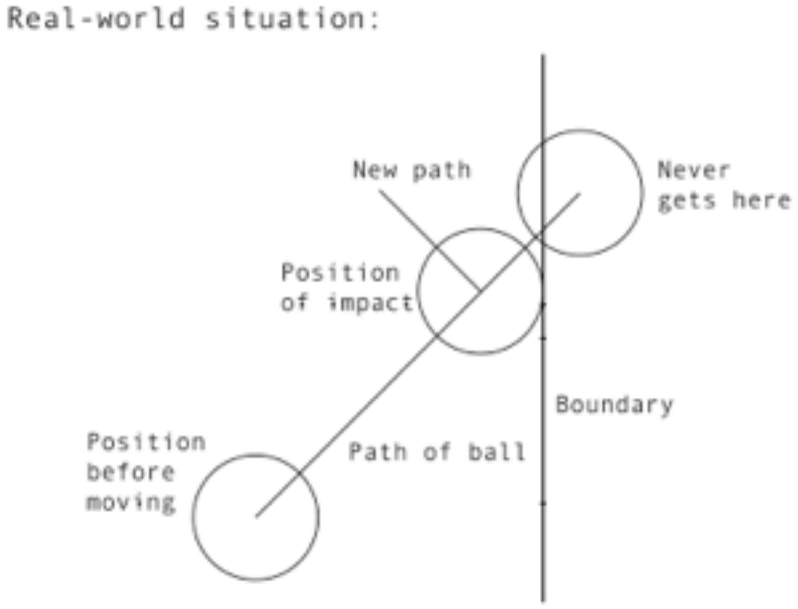
- Moving objects may meet other objects and boundaries
  - *Collision detection* algorithm detecting such situations
- Simple collision detection:
  - Width and/or height goes beyond some limit
    - » Bounds attributes in current example
    - » Bounds of nodes often available in high-level frameworks
- Potential problems:
  - Rounding errors may conceal collision event
  - Deep collisions:
    - » Collision is detected several times in sequence
    - » May lead to strange bouncing behaviour

Repositioning of Object on Collision

```
overShootHR = (self.posX+self.radius) - self.bHR
overShootHL = self.bHL - (self.posX-self.radius)
overShootVB = (self.posY+self.radius) - self.bVB
overShootVT = self.bVT - (self.posY-self.radius)
if overShootHR > 0:
   self.posX = self.posX - overShootHR
if overShootHL > 0:
   self.posX = self.posX + overShootHL
if overShootVB > 0:
   self.posY = self.posY - overShootVB
if overShootVT > 0:
   self.posY = self.posY + overShootVT
```

Model and Reality



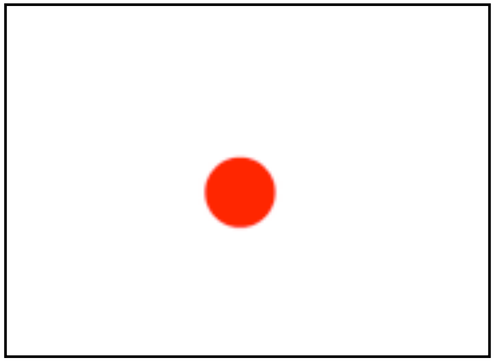


Simple Wall-to-Wall Bouncing

Bouncing always takes away some part of energy

- Use “bouncing factor”
- 1.0: No loss (unrealistic)
- In most cases use value smaller than 1, e.g. 0.9

```
if (overShootHR > 0) or (overShootHL > 0):
   self.vX = - self.vX*BOUNCE_LOSS
if (overShootVB > 0) or (overShootVT > 0):
   self.vY = - self.vY*BOUNCE_LOSS
```



Bounce and Friction

Surface absorbs some part of the energy and slows down the ball

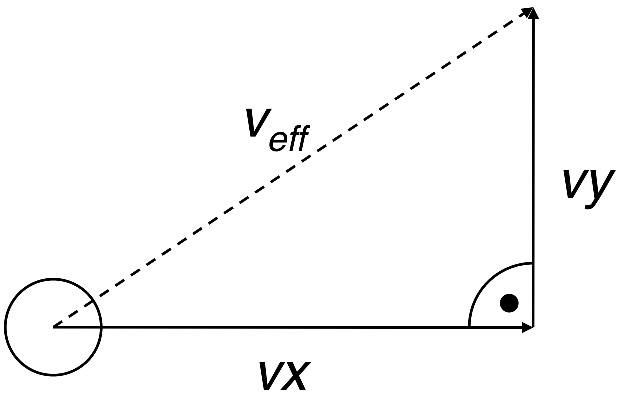
- Reduce velocity by some factor each frame
- Use “friction factor”
- 1.0: No friction (unrealistic)
- In most cases use value smaller than 1, e.g. 0.999

```
self.vX = self.vX*FRICT_LOSS
self.vY = self.vY*FRICT_LOSS
```

Minimum Speed

```
if self.getSpeed() < MIN_SPEED:
   self.vX = 0
   self.vY = 0
```

Needed for this: Effective speed out of x and y velocities



$$v_{eff} = \sqrt{vx^2 + vy^2}$$

```
def getSpeed(self):
   return math.sqrt(self.vX*self.vX+self.vY*self.vY)
```

Collision Detection Between Balls (1)

- Two moving balls may collide.
  - Collision detection needs access to data of both balls
  - Main program, or sprite manager class calling collision detection regularly
- Actual collision handling depends on geometry of objects
  - Bounding box is often not adequate for detection (e.g. for balls)

Main program loop:

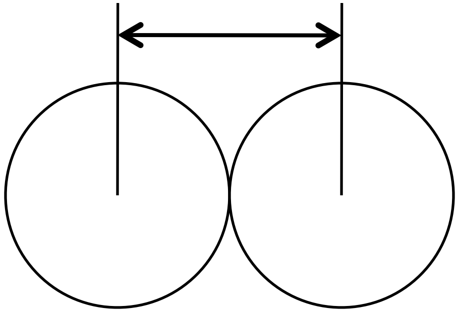
```
b1.update()
b1.draw(screen)
b2.update()
b2.draw(screen)
b1.handleCollision(b2)
```

Ball class:

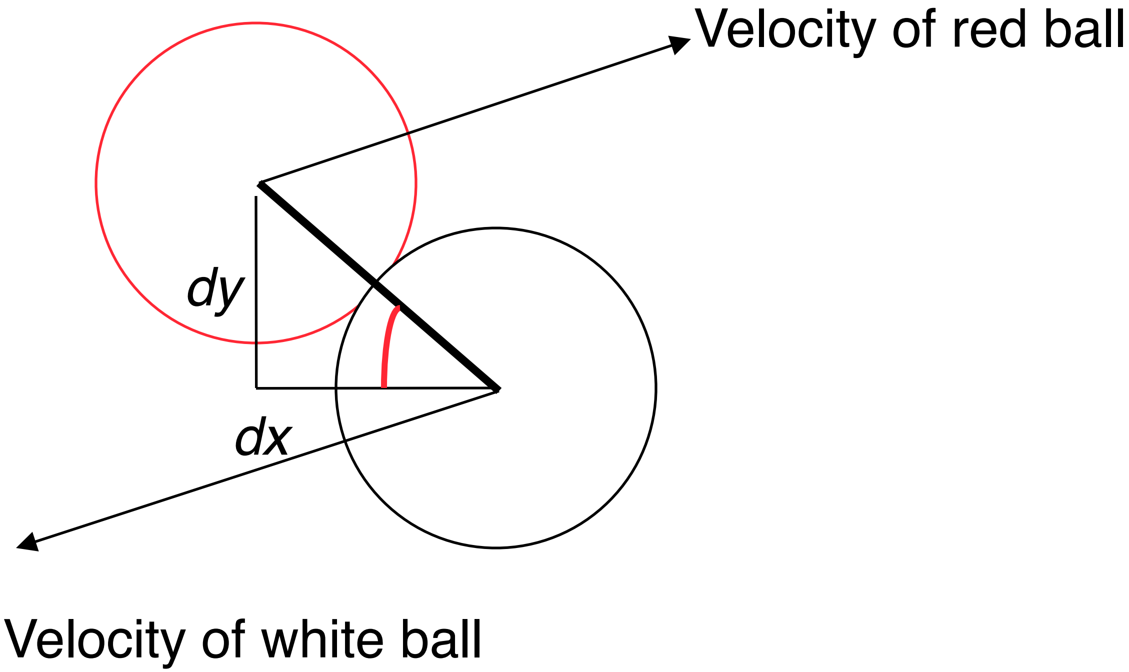
```
def handleCollision(self,other):
   …
```

Collision Detection Between Balls (2)

```
def handleCollision(self,other):
   dx = self.posX - other.getPosX()
   dy = self.posY - other.getPosY()
   dist = math.sqrt(dx*dx+dy*dy)
   overlap = self.radius + other.getRadius() - dist
   if overlap > 0:
      # Collision detected
      angle = math.atan2(dy,dx)
      cosa = math.cos(angle)
      sina = math.sin(angle)
      …
```

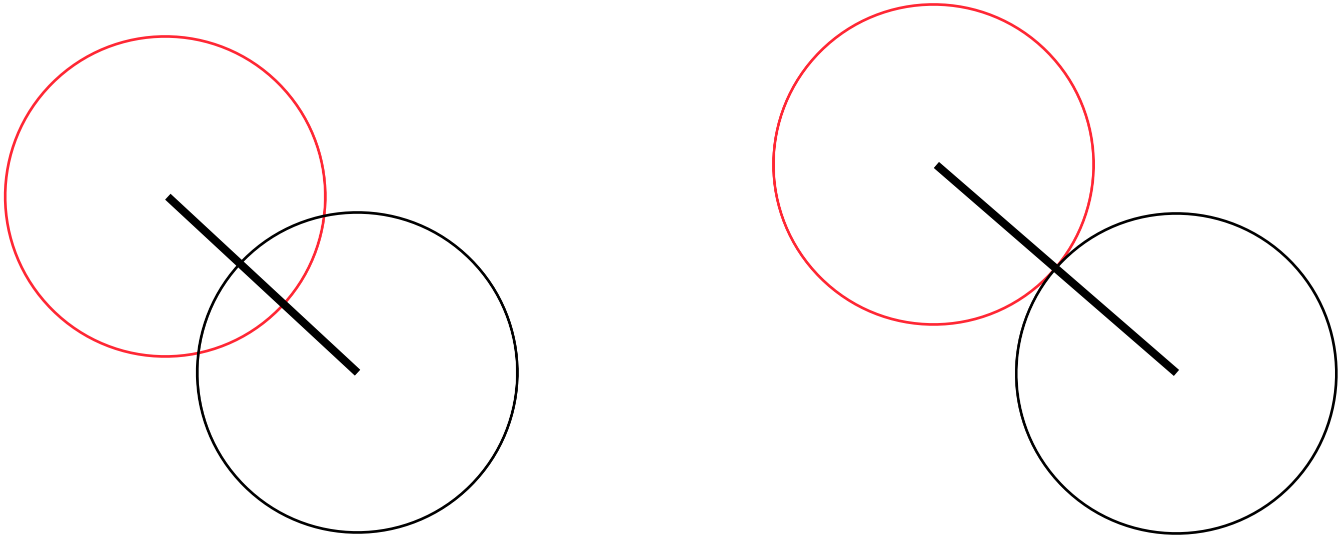


Collision Angle



```
angle = math.atan2(dy,dx)
```

Repositioning Balls At Collision Time

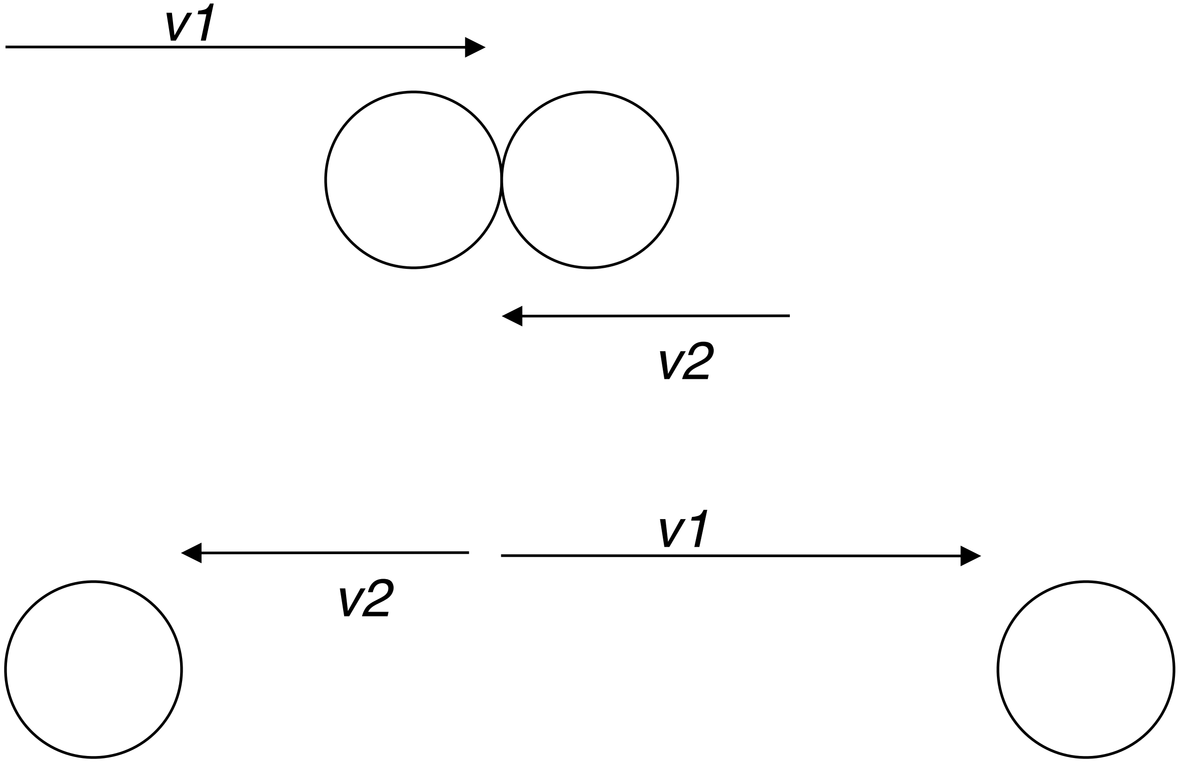


```
overlapX = overlap*cosa
overlapY = overlap*sina
posXNew = self.posX + overlapX/2
posYNew = self.posY + overlapY/2
otherPosXNew = other.getPosX() - overlapX/2
otherPosYNew = other.getPosY() - overlapY/2
self.setPos(posXNew,posYNew)
other.setPos(otherPosXNew,otherPosYNew)
```

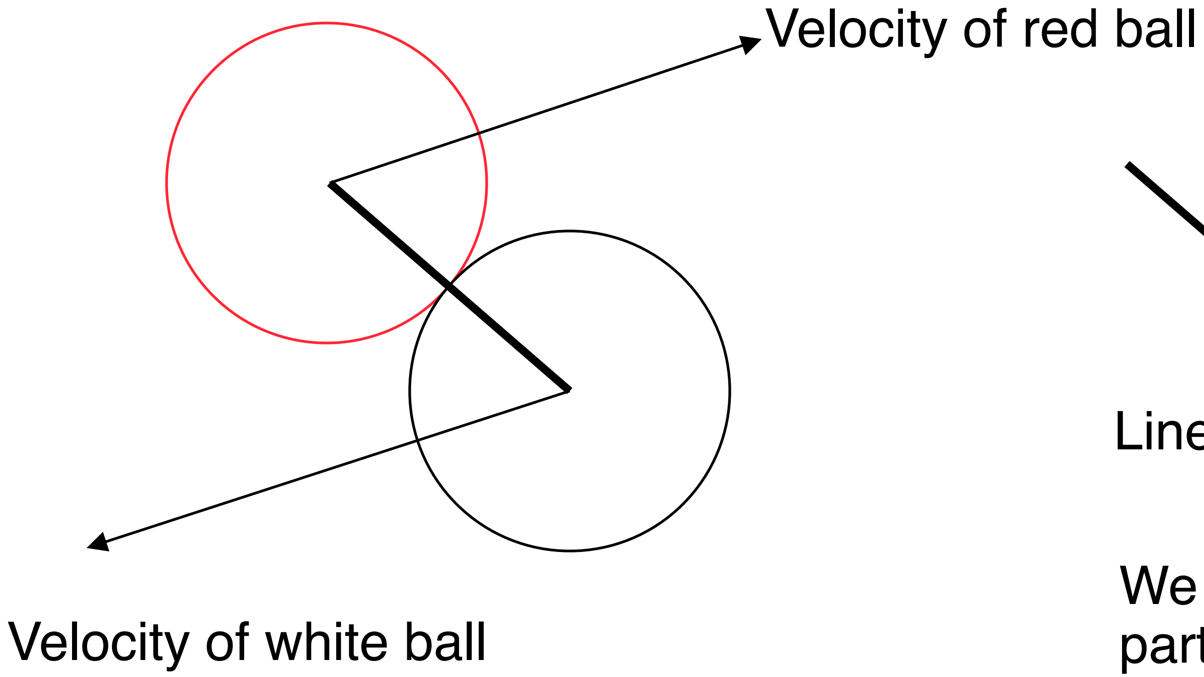
This is a simplification compared to the actual physical laws.

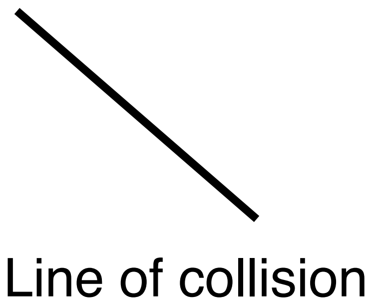
A Simple Case of Collision

- Two balls collide “head on”
- Balls have same size and same mass



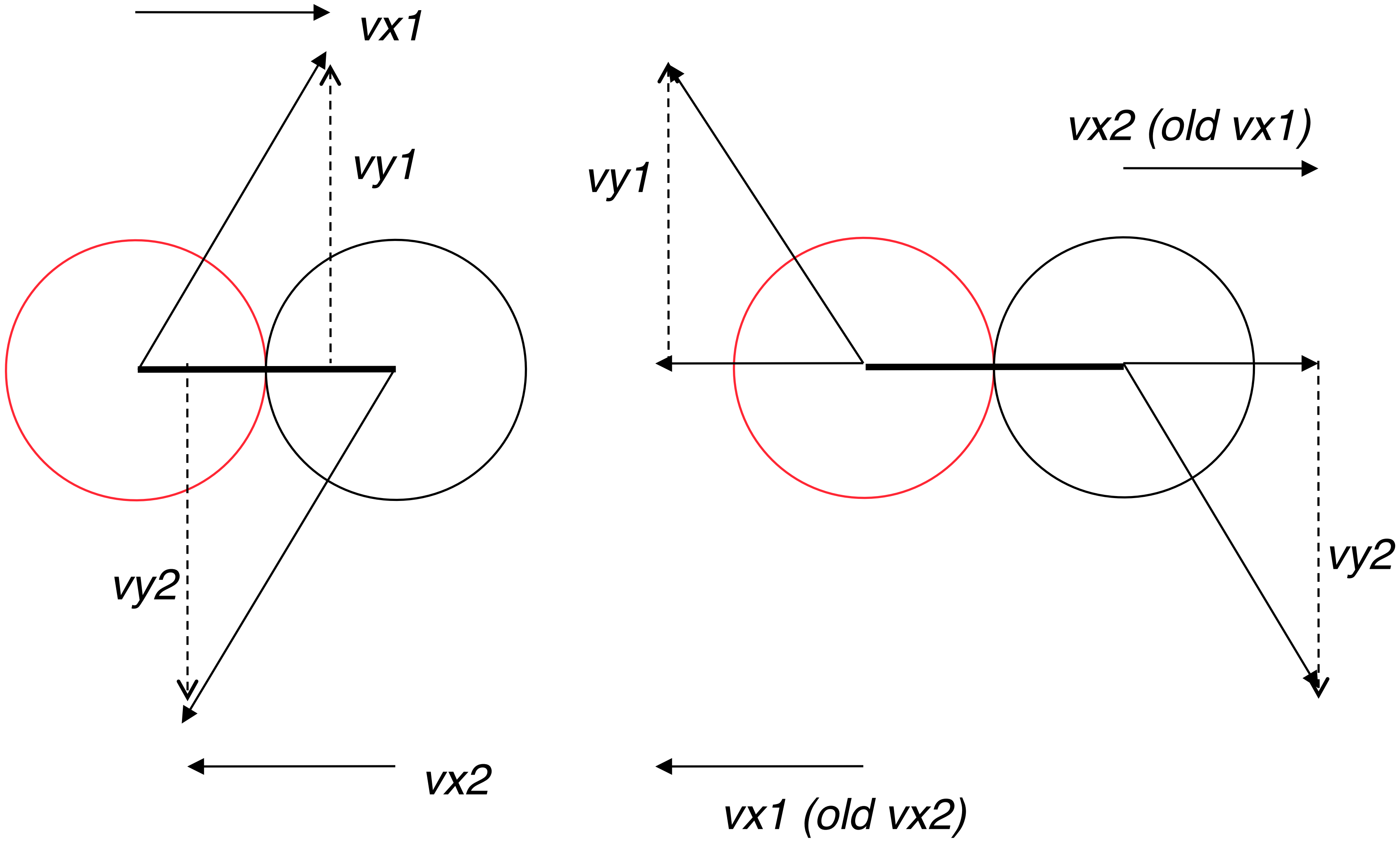
Physics of Collision, Step 1



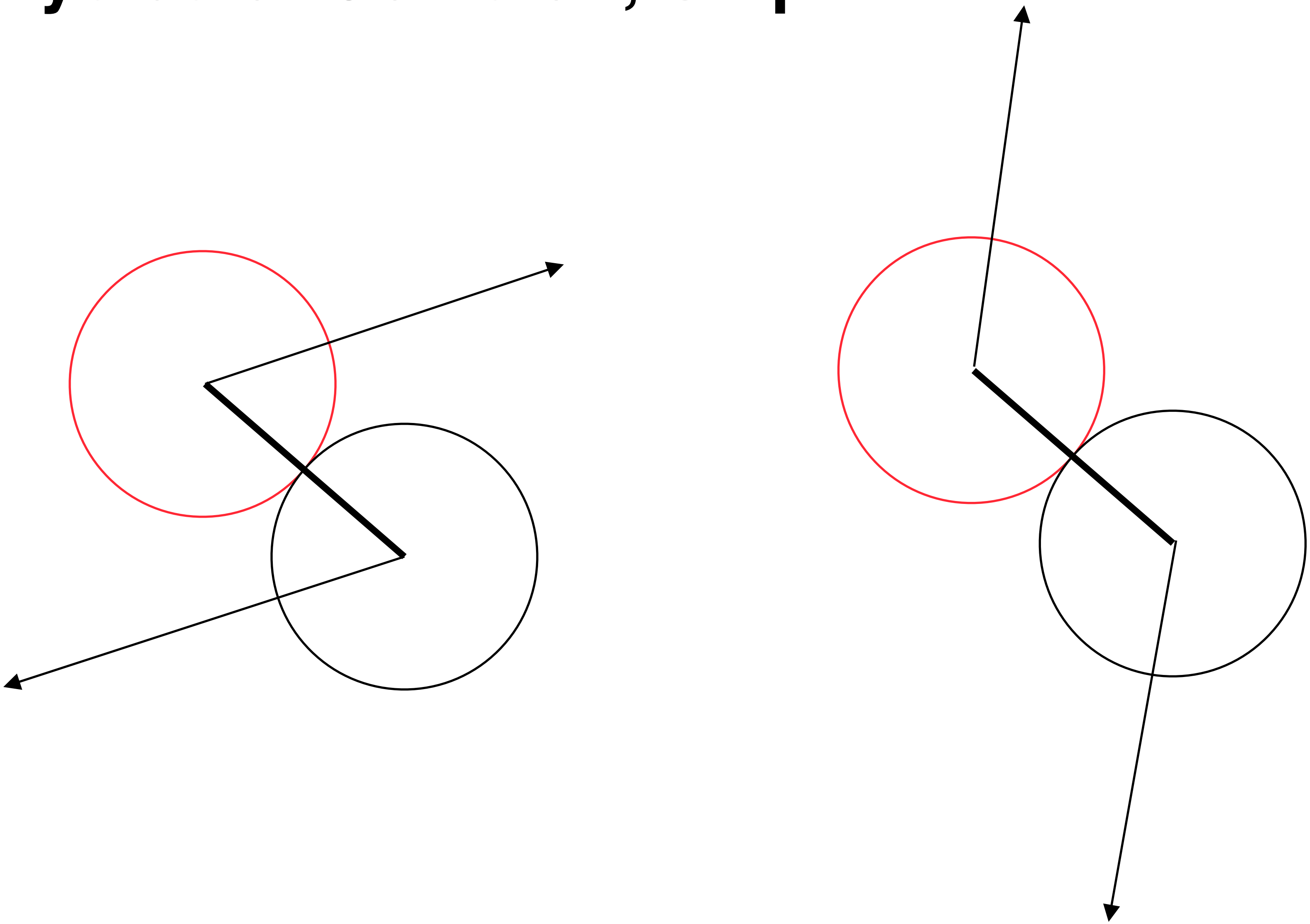


We need to determine those parts of the forces which actually contribute to the reaction, i.e. the projections on the collision line

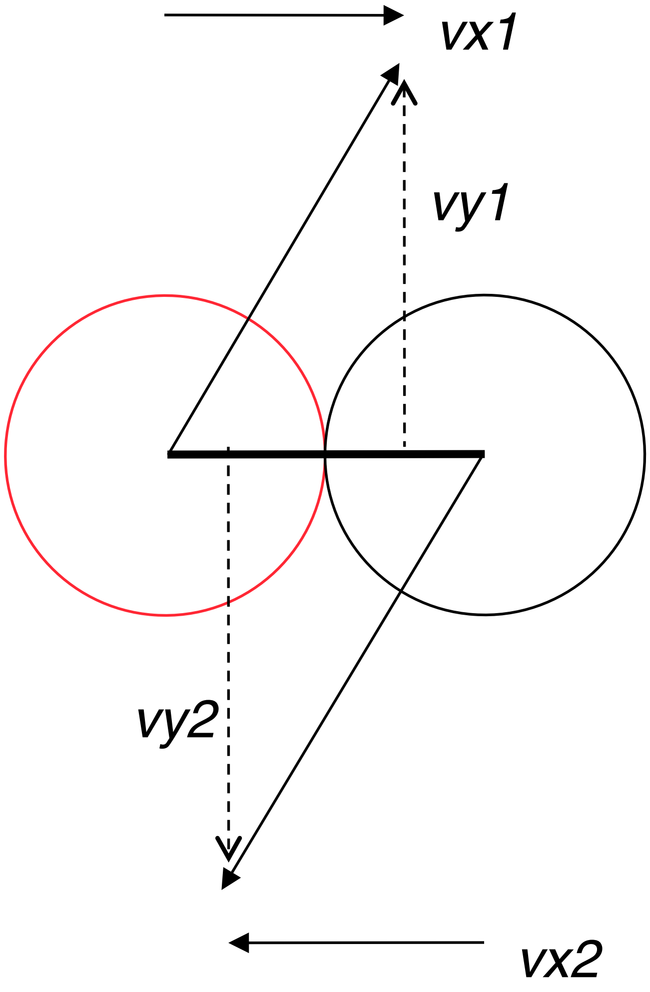
Physics of Collision, Step 2



Physics of Collision, Step 3



Computation Part 1



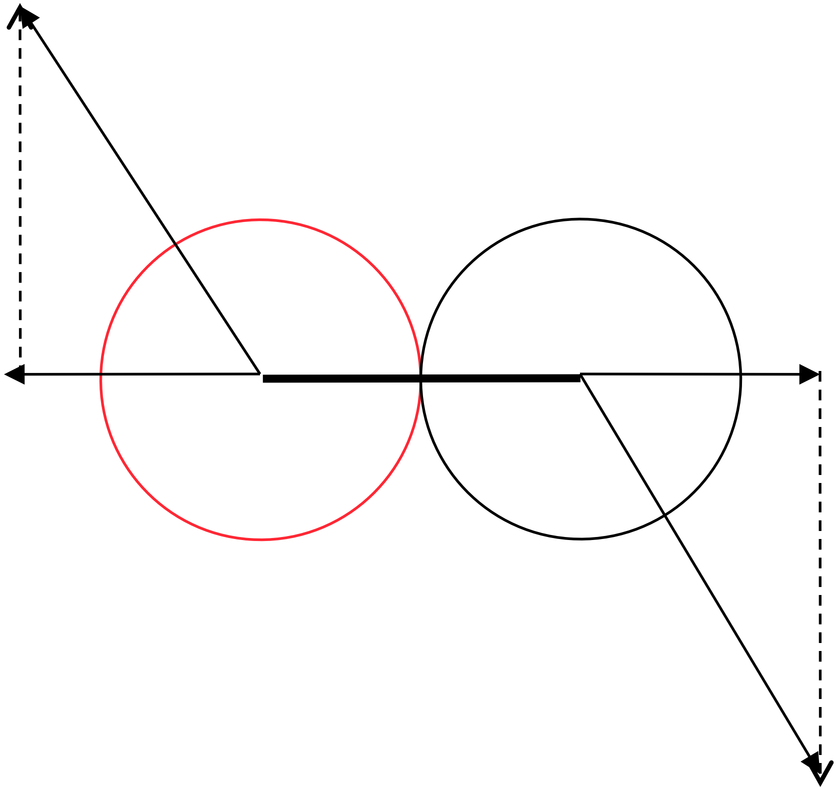
Counterclockwise rotation of vector (x, y):

$$x1 = \cos(\alpha) \cdot x + \sin(\alpha) \cdot y$$

$$y1 = \cos(\alpha) \cdot y - \sin(\alpha) \cdot x$$

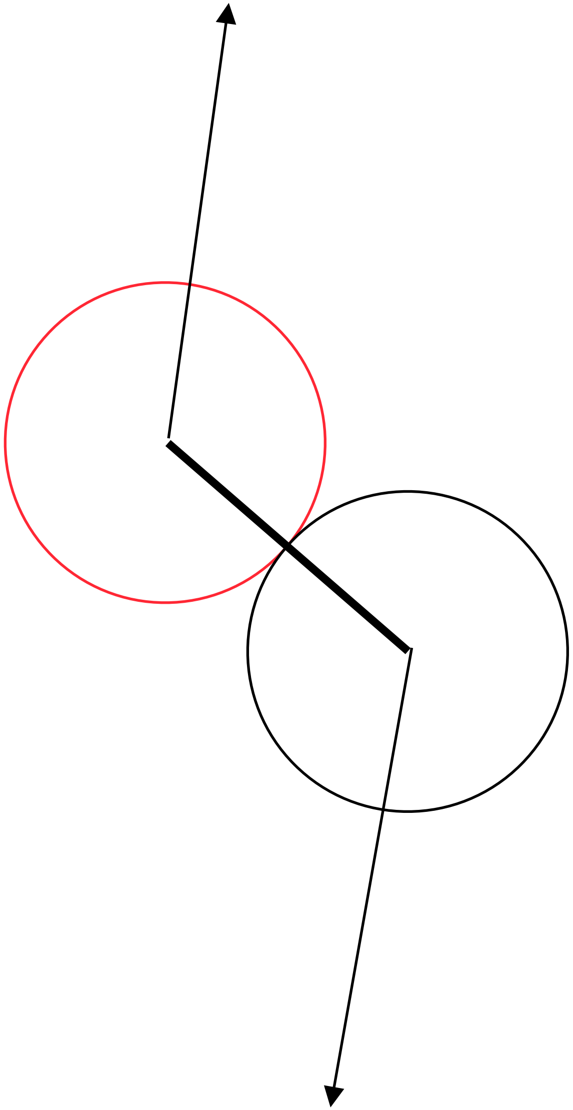
```
vx1 = self.vX
vy1 = self.vY
vx2 = other.getVelocityX()
vy2 = other.getVelocityY()
px1 = cosa*vx1 + sina*vy1
py1 = cosa*vx1 - sina*vy1
px2 = cosa*vx2 + sina*vy2
py2 = cosa*vx2 - sina*vy2
```

Computation Part 2



```
pNewx1 = px2
pNewy1 = py2
pNewx2 = px1
pNewy2 = py1
```

Computation Part 3



Clockwise rotation
of vector (x, y):

$$x1 = \cos(\alpha) \cdot x - \sin(\alpha) \cdot y$$

$$y1 = \cos(\alpha) \cdot y + \sin(\alpha) \cdot x$$

```
vXNew = cosa*pNewx1 - sina*pNewy1
vYNew = cosa*pNewx1 + sina*pNewy1
otherVXNew = cosa*pNewx2 - sina*pNewy2
otherVYNew = cosa*pNewx2 + sina*pNewy2
```

Final Step: Readjusting Velocities

- Taking into account energy loss through collision!

```
self.setVelocity(vXNew*BOUNCE_LOSS,vYNew*BOUNCE_LOSS)
other.setVelocity(otherVXNew*BOUNCE_LOSS,otherVYNew*BOUNCE_LOSS)
```

Physics: Speed, Velocity, Mass, Momentum

- *Speed:*
  - How fast is something moving (length/time)
- *Velocity:*
  - Vector describing movement: *speed* + *direction*
- *Mass:*
  - Basic property of object, depending on its material, leads under gravity to its *weight*
- *Momentum (dt. Impuls):*
  - Mass x Velocity
- *Principle of Conservation of Momentum (dt. Impulserhaltung):*
  - Total momentum of the two objects before the collision is equal to the total momentum after the collision.