Collision Detection

April 30, 2024

1 Simple Collision Detection

File as PDF

In particle simulations it is sometimes useful to detect collisions. This can be done for the sake of visualization or to really simulate physical collisions. Since we now know about classes. We will use this knowledge to imlement some basic collision detection to our particle system. The method we use for collision detection in this code is a simplified model and is not entirely physically correct. It's a basic form of collision detection and resolution that checks if two objects are overlapping and then pushes them apart.

In a more physically accurate model, you would also consider factors like:

- **Elasticity:** When two objects collide, they don't just stop or move away from each other. They bounce off each other with a certain amount of energy depending on their elasticity.
- **Momentum Conservation:** In a real-world collision, the total momentum of the system (the sum of the momenta of the two colliding objects) is conserved before and after the collision.
- **Angular Momentum:** If the collision is not head-on (i.e., the objects don't hit each other directly in the center), it can cause rotation.
- **Friction:** This could also play a role in the collision, depending on the surfaces of the colliding objects.

For the moment we would like to skip these physical complications and just implement a simple version. Later in the notebook you will get a more physically correct version.

[36]:
```python
import numpy as np
from time import sleep
import bqplot.pyplot as plt
from ipycanvas import Canvas, hold_canvas
import matplotlib.colors as mcolors
```

We actually split the task into a `solver class`, which is responsible for the time stepping through the animation and solving the collisions as well as `particle class` which represents the particle and holds all its properties.

The main ingredient for the solver is the solve_collision. This function looks for the overlap of two particles by calculating their distance. If the distance is smaller than the sum of their radii, then a collision occurs. To correct this collision, both particles involved in the colision are pushed back by the same amount delta*(distance_vec/dist). While this is only done for a pair always, it might create trouble for other pairs. Yet we ignore all that. Also note that this is not according to the momentum conservation, as both particle are pushed along the connecting line by the same amount.

[51]:
```
class Solver:
    gravity=np.array([0.,-10.])
    world_size=np.array([400.,400.])

    def __init__(self,dt,objects):
        self.dt=dt
        self.objects=objects

    def update(self,steps):
        self.sub_dt=self.dt/steps
        for i in range(steps):
            self.find_collisions()
            self.update_all(self.sub_dt)

    def find_collisions(self):
        for p1 in self.objects:
            for p2 in self.objects:
                if p1!=p2:
                    self.solve_collision(p1,p2)

    def solve_collision(self,p1,p2):
        distance_vec=p1.position-p2.position
        d=p1.radius+p2.radius
        distance2 =np.sum(distance_vec**2)
        if distance2<d**2 and distance2>0.:
            dist=np.sqrt(distance2)
            delta = 0.5*(d-dist)
            vec=delta*(distance_vec/dist)
            p1.position=p1.position+vec
            p2.position=p2.position-vec

    def update_all(self,dt):
        for object in self.objects:
            object.acceleration+=self.gravity
            object.update(dt)
            margin=10
            if object.position[0]>self.world_size[0]-margin:
                object.position[0]=self.world_size[0]-margin
            elif object.position[0]<margin:
                object.position[0]=margin
            if object.position[1]>self.world_size[1]-margin:
                object.position[1]=self.world_size[1]-margin
            elif object.position[1]<margin:
                object.position[1]=margin
```

The class below is the particle class, which holds all details of the particle and updates the particles position according to acceleration and speed. It ignores, however, all the collision, which are addressed by the Solver.

[52]:
```
class Particle:
    def __init__(self,R,p):
        self.position=p
        self.last_position=p
        self.acceleration=np.array([0.,0.])
        self.radius=R
        #self.velocity=v

    def set_position(self,position):
        self.position=position

    def update(self,dt):
        last_update_move=self.position-self.last_position
        new_position=self.position+last_update_move+self.acceleration*dt*dt
        self.last_position=self.position
        self.position=new_position
        self.acceleration=np.array([0.,0.])
```

Below we just generate a list of particles that we can supply to the solver.

[97]:
```
p=[Particle(10,np.array([400*np.random.rand(),400*np.random.rand()])) for i in␣
 ↪range(50)]
```

1.1 Initializing the simulation

This creates a solver object and initializes a list of positions for drawing.

[98]:
```
s=Solver(0.1,p)
x=[]
y=[]
for object in p:
    x.append(object.position[0])
    y.append(object.position[1])
```

1.2 Function to run the simulation

This holds our animation plotting with `ipywidgets`.

[99]:
```
def runsimIPC():
    for _ in range(150):
        s.update(4)
        with hold_canvas(c):
            c.clear()
            for object in p:
                c.fill_circle(object.position[0],400-object.position[1],object.
 ↪radius)
        sleep(0.02)
```

[100]:
```
c=Canvas(width=400,height=400)
c
```

[100]:
```
Canvas(height=400, width=400)
```

[101]:
```
runsimIPC()
```