PROJECT REPORT

IDEA :

The project comprises of three, simple and interactive, physical simulations built using Python and Pygame. These three simulations have been captioned Falling Balls, Discs on Table and Coalescence of Particles, respectively. One can understand the concepts used in the implementation of these three simultaions by refering to a 11^{th} standard Physics textbook.

TECHNOLOGY USED AND IMPLEMENTATION DETAILS :

As mentioned in the last head, the technology used in this project is limited to Python and Pygame. I'd like to elaborate on the implementation details of the project by interpreting the code of each of the three simulations one-by-one.

Starting off with balls.py, it nearly mirrors the behavior of balls falling through air, under the effect of gravity. The code begins with the importation of some python modules, namely, pygame, random, math and mixer modules. After which, the constructors of font and mixer classes have been called. Following this, I have mapped the background colour and dimensions of the pygame window to some values, defined a few objects, of the image, mixer and font classes (that help with loading the background image, including some sound files and defining the font type and size, respectively). Also, mass_of_air, friction, gravity and density(of balls) have been declared. I'd like to mention that here, gravity is not 9.8 m/s^2, it is a vector directed towards the negative y-axis, carrying a magnitude of 0.002 pixels, so is the case with many other physical quantities and athough this is not akin to what is known, it helps in portraying reality.
Moving on, I'd like to describe my first function; addVectors() takes two vectors as arguments (a vector is a tuple consisting of an angle and a magnitude), does vector addition and works out a new vector. The code is simple but it'll seem a bit confusing as the origin is on the top-left of the pygame window and all angles have been calculated with respect to the positive y-axis.
My second function is findBall(), it takes three arguments-the list of all Ball objects created, an x-coordinate, and a y-coordinate (these coordinates correspond to where one has clicked on the screen). The function iterates through the list and if distance of the mouse-click from centre of the Ball object turns out to be lesser than radius of the Ball object, it is implied that the Ball object has been 'selected' and is thus returned by the function. The purpose of findBall() will become clearer when it is called, later on.
Now, coming to collide(), which is a very important function, it takes two Ball objects as arguments, checks if the distance between their centres is lesser than the sum of their radii and if it is, reaches the conclusion that the Ball objects have collided and alters their speeds and directions accordingly, also, plays a sound on collision. It is as if the Ball objects are bouncing off a flat surface with the angle of the tangent (at their point of contact). To find out the new angle and speed

of each Ball object, addVectors() is called, the arguments of which are determined in a way so as to replicate the following formula :

$$v_1 = (m_1 – m_2 / m_1 + m_2) * u_1 + (2 * m_2 / m_1 + m_2) * u_2$$
$$v_2 = (m_2 – m_1 / m_1 + m_2) * u_2 + (2 * m_1 / m_1 + m_2) * u_1$$

(coefficient of elasticity (e) = 1)

The above formula has been derived from the equations given below :

$$m_1 * u_1 + m_2 * u_2 = m_1 * v_1 + m_2 * v_2 \text{ (conservation of momentum)}$$

$$1/2 * m_1 * u_1 * u_1 + 1/2 * m_2 * u_2 * u_2 = 1/2 * m_1 * v_1 * v_1 + 1/2 * m_2 * v_2 * v_2 \text{ (conservation of kinetic energy)}$$

After finding out the new speeds, they are multplied with friction, slowing them down a little; rest of the function works on solving the problem of internal bouncing (even after changing their angles and speeds, the Ball objects might still be overlapping because of drag, gravity and friction also influencing their movement...and if they still overlap then they will bounce back the way they came i.e. towards each-other, eventually getting trapped).
I'd now like to go over the class Ball, it's __init__() is used to create objects with attributes num, x, y, size, colour, thickness, speed, angle, mass and drag; display() is used to draw the Ball objects as circles and to label them with num, move() is responsible for the movement of Ball objects, it makes use of addVectors() to account for the effect of gravity, it also takes care of the drag experienced in air (which is proportional to the mass of the Ball object and obviously depends on its speed); lastly, bounce() makes sure the Ball object doesn't travel beyond the left, right and lower boundaries of the pygame window, it makes it bounce back, according to the laws of reflection (also, plays a sound and multiplies speed with friction).
Before creating objects of this class, the captioned pygame window is displayed and number_of_balls and an empty list named my_balls is declared. A for loop is used to create Ball objects which are appended to my_balls. It is here that the random module is primarily used, to assign values to object's attributes-size, x, speed and angle; mass is calculated by multiplying density to the square of size (this relates mass to size even though it's not the right kind of relation). Finally, a while loop is what keeps the simulation running until one decides to close it, this part of the code self-explanatory, it's where the various functions are called. Also, now, it's evident where and how findBall() is used, if a Ball object is 'selected' then the if-block makes sure that the Ball object travels with an appropriate speed and angle after we release it.

discs.py, my second simulation is not very different from balls.py, leaving out the graphics and audio, but of course, gravity has been ignored and so has the drag; reason behind creating this simulation was to make collision between the balls (here, discs) more observable, that's all.

coalesce.py is my third and last simulation, it has omitted many functions (like findBall() and collide()) there in balls.py and newly defined few of its own (like attract() and coalesce()). I'd only go over these newly defined functions as there's not much difference in this simulation compared to the first, apart from the graphics and audio obviously.

So, attract() takes two Particle objects as arguments and calculates the gravitational force between them, according to the following formula :

$$F = G * m_1 * m_2 / r * r$$

Here, gravitational constant is 0.2 instead of $6.674 * 10^{-11}$ $N.kg^{-2}.m^2$. It uses addVectors() to alter the speeds and angles of both the Particle objects (because of mutual gravitational attraction). coalesce() is responsible for the coalescence of Particle objects that collide, it uses the following formulas (pertaining to centre of mass) to reconstruct the new Particle object :

$$x = (m_1 * x_1 + m_2 * x_2) / m_1 + m_2$$

$$y = (m_1 * y_1 + m_2 * y_2) / m_1 + m_2$$

$$v = (m_1 * v_1 + m_2 * v_2) / m_1 + m_2$$

After, redefining particle1, particle2 is removed from the list of Particle objects.

IMAGES/SCREENSHOTS OR LINKS TO VIDEOS :

1. balls.py

2. discs.py

3. coalesce.py

FUTURE SCOPE OF MY PROJECT :

The simulations lack a scale i.e a parallel between pixels and SI units, which can be worked on. Also, concepts of rotation can be applied to make the simulations more realistic. The third simulation has many applications, it can be modified to build a star formation, solar system or soap bubbles simulation.

HOW WAS MY OVERALL EXPERIENCE WHILE DOING THE PROJECT :

Honestly, doing the project wasn't very difficult because I closely followed the tutorial http://www.petercollingridge.co.uk/tutorials/pygame-physics-simulation/. I learn't a lot over the course of this project but I regret not doing much beyond what the tutorial taught me. I'll keep this in mind the next time I do a project and try to think more differently.