Project Report: Physics Simulation with Pygame

1. Introduction

The purpose of this project report is to provide a detailed overview of a physics simulation developed using the Pygame framework. Pygame is a Python library commonly used for game development but can also be applied to create physics simulations. This report will cover the frameworks used, simulation logic, techniques employed, advantages and disadvantages, definitions of key terms, as well as a summary, conclusion, and suggestions for further improvements.

2. Frameworks and Tools Used

The physics simulation project utilizes the following frameworks and tools:

- Pygame: A Python library that provides functionalities for graphics rendering, user input handling, and game logic, which are essential for building a physics simulator.

- Python: The programming language used for developing the simulation.

- Integrated Development Environment (IDE): Visual Studio Code and Python IDLE for coding and testing the simulation.

- Graphics Editing Software: Tools like Adobe Photoshop, Canva, and 3D Paint for creating or editing the simulation's visual elements.

3. Simulation Logic and Features

The physics simulation developed with Pygame incorporates the following logic and features:

- Loading Screens: The `loadscreen.py` file implements loading screens to provide visual feedback on the initialization process of the simulation. This enhances the user experience and ensures the smooth loading of necessary assets.

- Loading Animation: The `loadscreen.py` file also includes a loading animation that visually represents the progress of loading. This animation creates a sense of anticipation and engagement for the user.

- Vector Visualization: The `vectors.py` file includes a function that displays the velocity vector of a simulated object on the screen. This feature allows users to visualize the motion and direction of objects within the simulation.

4. Techniques Employed

The physics simulation project employs the following techniques:

- Graphics Rendering: Pygame's graphics capabilities are utilized to render objects, backgrounds, and visual elements on the screen. This enables the representation of simulated objects and their interactions in a visually appealing manner.

- User Input Handling: Pygame provides functionality for handling user input, such as keyboard or mouse events. This allows users to interact with the simulation, manipulate objects, and observe the effects of their actions.

- Physics Simulation: The simulation logic applies fundamental physics principles, such as Newton's laws of motion and principles of kinematics, to calculate and simulate the behavior of objects within the virtual environment. This enables realistic object interactions and motion within the simulation.

5. Advantages and Disadvantages

Advantages:

- Pygame provides a convenient and accessible framework for building physics simulations, thanks to its wide range of functionalities.

- Python's simplicity and readability make it suitable for developing and maintaining complex physics simulations.

- Pygame's built-in functionalities for graphics rendering and user input handling significantly reduce the development time and effort required.

- The Python ecosystem offers extensive libraries and resources that can be leveraged to enhance the simulation, such as NumPy for mathematical computations or Matplotlib for data visualization.

Disadvantages:

- Pygame may not offer the same level of performance as more specialized physics simulation frameworks.

- Implementing complex physics models or advanced graphical effects may require additional libraries or custom implementations.

- Pygame's documentation and community resources, while helpful, may not be as extensive as those available for more popular physics simulation frameworks.

6. Definitions

- Pygame: A Python library commonly used for game development can also be employed to build physics simulations.

- Physics Simulation: A virtual environment that applies physical laws and principles to simulate the behavior and interactions of objects.

- User Input: Actions performed by the user, such as providing initial conditions, applying forces, or observing the simulation results.

- Vector Visualization: The graphical representation of vectors, such as velocity or acceleration, to visually convey information about object motion

.

7. Summary

The physics simulation project developed with Pygame demonstrates the capability of creating interactive and visually appealing simulations. By utilizing Pygame's functionalities, the project provides an immersive experience for users to explore and observe the behavior of simulated objects in a virtual environment. The loading screens and animations enhance the user experience during the initialization phase, while the vector visualization feature provides valuable insights into object motion.

8. Conclusion

In conclusion, the physics simulation project showcases the potential of Pygame as a framework for developing physics simulations. With Pygame's extensive functionalities and Python's simplicity, developers can create engaging and educational simulations that facilitate the understanding of physical concepts. While Pygame may have certain limitations compared to specialized physics simulation frameworks, it offers accessibility and ease of use, particularly for beginners and developers familiar with Python.

9. Further Scope for Improvements

To further enhance the physics simulation project, the following improvements can be considered:

- Advanced Physics Models: Implementing more sophisticated physics models, such as fluid dynamics or rigid body dynamics, to simulate a wider range of physical phenomena.

- Real-Time Data Visualization: Incorporating real-time data visualization techniques to display and analyze simulation data, allowing users to observe and interpret the results more effectively.

- Interactive Controls: Adding interactive controls that enable users to manipulate simulation parameters in real-time, offering a more engaging and interactive experience.

- Multiple Simulations: Allowing users to create and compare multiple simulations simultaneously, facilitating comparative analysis and exploration of different scenarios.

- Export and Sharing: Providing options to export simulation data or share simulations with others, enabling collaboration and educational purposes.

By implementing these improvements, the physics simulation project can become more comprehensive, visually appealing, and educational, offering a broader range of simulations and interactive features.

Overall, the physics simulation project demonstrates the potential of Pygame for creating interactive and visually engaging simulations. It serves as a foundation for further exploration and development in the field of physics simulations using Python.

Pseudocodes of each file:

loadScreen.py

loadscreen.py provides loading screen functionality before the simulation starts. It displays a series of loading screens to engage the user and prepare the environment for the simulation. Users have the option to proceed or exit the loading screens.

Start

|

V

Initialize Pygame

|

V

Create Clock Object

|

V

Create Screen Surface (size: window\_width x window\_height)

|

V

Define Constants (window\_height, window\_width, resize)

|

V

Define load\_screen Function

|

V

Load Image File 'lo1.png' and Create Surface Object (surface)

|

V

Blit Surface on Screen at Position (0, 0)

|

V

Update Display

|

V

Check Events

|

V

Check for Quit Event

|

V

Check for Key Press Event (K\_k)

|

V

Return True to Continue Loading Screens

|

V

Check Events

|

V

Check for Quit Event

|

V

End if Quit Event Detected

|

V

End if K\_k Key Press Detected

|

V

Return False to Stop Loading Screens

|

V

End

main.py

In main.py, the core functionality of the simulator is implemented. It handles the initialization of the simulation environment, including the creation of the display window using the Pygame library. The main loop continuously updates the positions of objects based on their velocities, applies forces such as gravity and friction, detects collisions, and renders the objects on the screen. Users can interact with the simulation by adding or removing objects, adjusting their properties, and observing their motion in real-time.

Start

|

V

Import Required Modules (pygame, sys, math)

|

V

Initialize Pygame

|

V

Create Clock Object

|

V

Create Screen Surface (size: 1000 x 650)

|

V

Create Font Object (text\_font)

|

V

Define net\_color (White)

|

V

Define Ball Class

|

V

Define Constructor (\_\_init\_\_) for Ball Class

|

V

Define update Method for Ball Class

|

V

Define draw Method for Ball Class

|

V

Define create\_balls Function

|

V

Create Empty List to Store Ball Objects (balls)

|

V

Loop Over a Range (num\_balls) to Create Ball Objects

|

V

Add Ball Object to balls List

|

V

Define handle\_collisions Function

|

V

Loop Over Balls in balls List

|

V

Check for Collision with Other Balls

|

V

Update Velocity and Position Based on Collision

|

V

Define main Function

|

V

Set Number of Balls (num\_balls)

|

V

Call create\_balls Function to Generate Ball Objects

|

V

Create a Variable to Control the Main Loop (running = True)

|

V

Start Main Loop

|

V

Check Events

|

V

Check for Quit Event

|

V

End if Quit Event Detected

|

V

Clear Screen

|

V

Call handle\_collisions Function to Handle Ball Collisions

|

V

Loop Over Balls in balls List

|

V

Call Update Method for Each Ball

|

V

Loop Over Balls in balls List

|

V

Call Draw Method for Each Ball

|

V

Update Display

|

V

End Main Loop

|

V

End

Vectors.py

vectors.py focuses on displaying the velocity vectors of objects in the simulation. It calculates the magnitude and direction of the velocity vector and visualizes it on the screen as a line originating from the object's position. Users can observe the speed and direction of objects' motion by analyzing the displayed velocity vectors.

Start

|

V

Initialize Pygame

|

V

Create Clock Object

|

V

Create Screen Surface (size: 1000 x 650)

|

V

Create Font Object (text\_font)

|

V

Define net\_color (White)

|

V

Define show\_vectors Function

|

V

Check Events

|

V

Check for Quit Event

|

V

Create Velocity Text Surface (net\_surface) with Text: Magnitude of Velocity (sqrt(v\_x^2 + v\_y^2))

|

V

Create Rectangle for Velocity Text (net\_rect) at Center of Ball

|

V

Blit Velocity Text Surface on Screen

|

V

Draw Velocity Vector Line on Screen from Ball Center to (ball\_center[0] - 2\*v\_x, ball\_center[1] - 2\*v\_y)

|

V

Update Display

|

V

End if Quit Event Detected

|

V

End

Code files:

loadScreen.py

import pygame

from sys import exit

import random

pygame.init()

clock = pygame.time.Clock()

window\_height = 650

window\_width = 1259

screen = pygame.display.set\_mode((window\_width,window\_height), pygame.FULLSCREEN)

screen.fill("Black")

clock.tick(1)

loader = [1, 9, 27, 5, 0]

def load\_screen(screen):

    """

    Displays a series of loading screens before the game starts.

    Args:

        screen (pygame.Surface): The screen to display the loading screens on.

    Returns:

        bool: True if the loading screens should continue, False if they should stop.

    """

    global window\_height

    global window\_width

    global resize

    surface  = pygame.image.load('graphics/Scenes/lo1.png').convert()

    screen.blit(surface, (0,0))

    clock.tick(5)

    pygame.display.update()

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()

            exit()

        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K\_k:

                return False

    return True

bar = pygame.image.load('graphics/Scenes/lobar.png').convert\_alpha()

barRect = bar.get\_rect(topright = (0,0))

def loading(screen):

    """

    Displays a loading animation on the screen.

    Args:

        screen (pygame.Surface): The screen to display the loading animation on.

    Returns:

        bool: True if the loading animation should continue, False if it should stop.

    """

    global window\_height

    global loader

    bar = pygame.image.load('graphics/Scenes/lobar.png').convert\_alpha()

    surface  = pygame.image.load('graphics/Scenes/lo1.png').convert()

    surface2  = pygame.image.load('graphics/Scenes/lo2.png').convert\_alpha()

    if barRect.right>1133:

        loader[4]=loader[3]

    elif barRect.right>378:

        loader[4]=loader[2]

    elif barRect.right>315:

        loader[4]=loader[1]

    elif barRect.right>126:

        loader[4]=loader[0]

    else:

        loader[4] = 200

    if barRect.right<=1259:

        barRect.right += 0.5\*loader[4]

    else:

        return False

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()

            exit()

    screen.blit(surface, (0,0))

    screen.blit(bar, barRect.topleft)

    screen.blit(surface2, (0,0))

    pygame.display.update()

    clock.tick(20)

    return True

vectors.py

import pygame

from sys import exit

from math import sqrt

pygame.init()

clock = pygame.time.Clock()

screen = pygame.display.set\_mode((1000,650), pygame.FULLSCREEN)

text\_font = pygame.font.Font(None ,50)

net\_color = "White"

def show\_vectors(screen, body):

    """

    Displays the velocity vector of a body on the screen.

    Args:

        screen (pygame.Surface): The screen to display the vector on.

        body (Ball): The body for which to display the vector.

    """

    for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                exit()

    net\_surface = text\_font.render(f'{round(sqrt((body.v\_x\*\*2)+(body.v\_y\*\*2)))}', False, 'White')

    net\_rect = net\_surface.get\_rect(center = body.ball\_rect.center)

    screen.blit(net\_surface, (net\_rect.topright[0] + 50, net\_rect.topright[1]))

    pygame.draw.line(screen, net\_color, (body.ball\_rect.center), (body.ball\_rect.center[0] - 2\*body.v\_x,body.ball\_rect.center[1] - 2\*body.v\_y\*-1), 3)

main.py

#importing Modules

import pygame

from sys import exit #To close the window

import random #To genrate random variables

import loadScreen

import vectors

#Initializing PyGame

pygame.init()

#Screen Setup

window\_height = 650

window\_width = 1259

screen = pygame.display.set\_mode((window\_width, window\_height), pygame.FULLSCREEN) #Makes a window

pygame.display.set\_caption('Float Ball Bouncing') #Sets the name of the window

clock = pygame.time.Clock() #To add time delay in game loop

#Making required surfaces

bg\_surface = pygame.image.load('graphics/bg/bg.png').convert()

bg\_grd = pygame.image.load('graphics/bg/bground.png').convert\_alpha()

bg\_water = pygame.image.load('graphics/water/water.png').convert()

bg\_moon = pygame.image.load('graphics/moon/moonsky.png').convert()

bg\_moongrd = pygame.image.load('graphics/moon/moongrd.png').convert\_alpha()

midscene = pygame.image.load('graphics/Scenes/enter2.png').convert()

legend = pygame.image.load(('graphics/Scenes/legends.png')).convert\_alpha()

#Ball class to store and control properties of body aka ball

class Ball:

    t=0.1

    balls = []

    def \_\_init\_\_(self, x, y, g):

        self.x = x

        self.y = y

        self.e = 0.6 #Coefficient of restitution

        self.g = g

        self.gNet = g

        self.v\_x = 0

        self.v\_y = 0

        self.f = 0.001 #Drag Coefficient

        self.density = 1

        self.volume = 0.08

        self.mass = self.density\*self.volume

        self.ball\_surface = pygame.image.load('graphics/ball/1.png').convert\_alpha()

        self.ball\_rect = self.ball\_surface.get\_rect(midbottom = (self.x,self.y))

        self.dragging = False

        Ball.balls.append(self)

    def drag(self):

         #Assuming drag directly proportional to -kv

         self.v\_x += -self.v\_x\*self.f

         self.v\_y += -self.v\_y\*self.f

    def update\_position(self):

            disp\_y = (self.v\_y\*Ball.t+0.5\*(self.gNet)\*Ball.t\*\*2)\*20  #s = ut + 0.5at\*\*2

            self.v\_y += (self.gNet)\*Ball.t  #v=u+at

            self.ball\_rect.bottom += disp\_y #updates the coordinates of the bottom most point

            self.y = self.ball\_rect.bottom #the update coordinates are stored in y value

            if self.y < 0:

                self.remove\_ball()

    def replot\_x(self,scale):

        self.ball\_rect.left \*= scale

    def remove\_ball(self):

        if len(Ball.balls)>0:

            Ball.balls.remove(self) #Remove ball from the balls list in Ball class

    def sense\_medium(self, density):

        #To determine the surrounding medium of the body

        for i in range(len(Ball.balls)): #To update gNet of each ball with the value defined of g defined in back\_dict for different scenes

            Ball.balls[i].gNet = back\_dict[back]

        if back == 1:

            #Calculating buoyant force due to medium

            #Assumed ball to be cube

            fb = 0

            for i in Ball.balls:

                if i.ball\_rect.bottom>280:

                    if 280<i.ball\_rect.bottom<=380:

                        fb = density\*self.g\*(self.ball\_rect.bottom-280)\*self.volume/(self.mass\*100)

                        i.f = 0.01

                    else:

                        fb = density\*self.g\*self.volume/self.mass

                        i.f = 0.07 #If the ball is completely inside water then drag coefficeint is set to 0.07

                    i.gNet = i.g - fb

                else:

                    i.gNet = i.g

                    i.f = 0.01

#Turning on and off Interactive Mode

interactive = False

curr\_ball = None

ball1 = Ball(window\_width//2, window\_height-130, 10) #Creating a Ball object

back = 0 #To count current scene

back\_dict = {0:10, 1:1, 2:1.6} #Stores value of acceleration due to gravitation force.

#In back\_dict the value for key = 1 is density of the medium rather than acceleration value

c\_pressed = False #To check if c key is pressed.

mousebutton = False

#Font generation

text\_font = pygame.font.Font("graphics/font/gooddog-plain.regular.ttf",40)

#Showing waiting screen

while loadScreen.load\_screen(screen):

    continue

#Showing loading Screen

while loadScreen.loading(screen):

    continue

size = screen.get\_size()

scale = round(size[1]/window\_height, 2)

bottomline = window\_height-130\*scale #To set the collision point

if size[0]!=window\_width:

    bg\_surface = pygame.transform.scale\_by(bg\_surface, scale)

    bg\_grd = pygame.transform.scale\_by(bg\_grd, scale)

    bg\_water = pygame.transform.scale\_by(bg\_water, scale)

    bg\_moon = pygame.transform.scale\_by(bg\_moon, scale)

    bg\_moongrd = pygame.transform.scale\_by(bg\_moongrd, scale)

    midscene = pygame.transform.scale\_by(midscene, scale)

    for i in Ball.balls:

        i.replot\_x(scale)

    window\_height = size[1]

    window\_width = size[0]

    screen = pygame.display.set\_mode(size, pygame.FULLSCREEN)

#Main loop

while True:

    #Rendering the text to show density of the medium

    add\_surface = text\_font.render(f'Density of medium = {(round(back\_dict[1], 2))}', None, 'Black')

    add\_rect = add\_surface.get\_rect(topleft = (25,25))

    #To detect any event taking place

    for event in pygame.event.get():

        #Quit and close the window if Close Button(X) is pressed

        if event.type == pygame.QUIT:

            pygame.quit()

            exit()

        elif event.type == pygame.VIDEORESIZE:

            scale = round(event.h/window\_height, 2)

            for i in Ball.balls:

                i.replot\_x(scale)

            window\_height = event.h

            window\_width = event.w

            screen = pygame.display.set\_mode((window\_width, window\_height), pygame.RESIZABLE)

            print(scale)

            bg\_surface = pygame.transform.scale\_by(bg\_surface, scale)

            bg\_grd = pygame.transform.scale\_by(bg\_grd, scale)

            bg\_water = pygame.transform.scale\_by(bg\_water, scale)

            bg\_moon = pygame.transform.scale\_by(bg\_moon, scale)

            bg\_moongrd = pygame.transform.scale\_by(bg\_moongrd, scale)

            midscene = pygame.transform.scale\_by(midscene, scale)

        elif event.type == pygame.KEYDOWN:

             #Changing the y velocity of ball when Space key is pressed

             for i in Ball.balls:

                if event.key == pygame.K\_SPACE and abs(i.v\_y)<5 and (i.ball\_rect.bottom > window\_height-round(135\*scale) and back == 0):

                    i.v\_y = -random.randint(10, 20)

                if event.key == pygame.K\_SPACE and (abs(i.v\_y)<=1 and back == 1):

                    i.v\_y = -random.randint(6, 10)

                if event.key == pygame.K\_SPACE and abs(i.v\_y)<5 and (i.ball\_rect.bottom > window\_height-round(160\*scale) and back == 2):

                    i.v\_y = -random.randint(6, 9)

             #Adding new balls

             if event.key == pygame.K\_a and len(Ball.balls)<3 : #Limits the maximum number of balls to 3

                #Adding balls at different position depending on the scene

                if back == 0:

                    ball = Ball(random.randint(50, 900), bottomline, 10)

                elif back == 1:

                    ball = Ball(random.randint(50, 900), bottomline, 10)

                else:

                    ball = Ball(random.randint(50, 900), 0, 10)

             #Removing balls based on FIFO (First-in-first-out)

             elif event.key == pygame.K\_r:

                    if len(Ball.balls)>0:

                        Ball.balls[0].remove\_ball()

             #Changing the scenes

             elif event.key == pygame.K\_c:

                 if not c\_pressed: #If c\_pressed == False, value of back is updated to change scene

                    if back<2:

                        back+=1

                    else:

                        back -=2

                    c\_pressed = True

             if event.key == pygame.K\_RETURN and c\_pressed: #Checks if Enter key is pressed after c

                 c\_pressed = False

             #To change the density of medium by RIGHT and LEFT arrow keys

             if event.key == pygame.K\_RIGHT and back == 1:

                 if 0.8<=back\_dict[back]<1.5: #Restricts the values to be between 0.8 and 1.5

                    back\_dict[back]+=0.1

             if event.key == pygame.K\_LEFT and back == 1:

                 if 0.8<back\_dict[back]<=1.6:

                    back\_dict[back]-=0.1

             #Code for interactive mode

             if event.key == pygame.K\_i:

                 interactive = not interactive

        elif event.type == pygame.MOUSEBUTTONDOWN and interactive:

             mousebutton = True

             for i in Ball.balls:

                  if i.ball\_rect.collidepoint(event.pos):

                       i.dragging = True

                       curr\_ball = i

        elif event.type == pygame.MOUSEBUTTONUP or interactive == False:

             for i in Ball.balls:

                  i.dragging = False

             mousebutton = False

             curr\_ball = None

        if interactive:

            if event.type == pygame.MOUSEBUTTONDOWN and len(Ball.balls) < 3:

                dragger = True

                for i in Ball.balls:

                        dragger = dragger and i.dragging

                if not dragger or len(Ball.balls) == 0:

                    if 50<=event.pos[0]<=window\_width-50 and 50<=event.pos[1]<=bottomline-50:

                        ball = Ball(event.pos[0], event.pos[1], 10)

    if interactive:

        if mousebutton and curr\_ball != None:

            curr\_ball.ball\_rect.center = pygame.mouse.get\_pos()

            curr\_ball.v\_y = 0

    screen.blit(bg\_surface, (0,0)) #Adds the background on the screen

    if c\_pressed: #If c\_pressed == True all the balls are removed and "Press Enter is shown"

        for i in Ball.balls:

            i.remove\_ball()

        screen.blit(midscene, (0,0))

    else:

    #Else the background is added depending on current scene

    #Also sets the bottomline of each scene

        if back == 1:

            screen.blit(bg\_water, (0,0))

            screen.blit(add\_surface, add\_rect)

            bottomline = window\_height

        elif back == 0:

            screen.blit(bg\_surface, (0,0))

            screen.blit(bg\_grd, (0, 0))

            bottomline = window\_height- round(130\*scale)

        elif back == 2:

            screen.blit(bg\_moon, (0,0))

            screen.blit(bg\_moongrd, (0,0))

            bottomline = window\_height-round(155\*scale)

        screen.blit(legend, (window\_width-150, 0)) #To add legends on screen

    #Basic Physics and mechanics

    for i in Ball.balls:

            screen.blit(i.ball\_surface, i.ball\_rect.topleft) #Shows ball on screen

            vectors.show\_vectors(screen, i) #To show vectors of each ball

            i.sense\_medium(back\_dict[back]) #Tracks medium

            i.drag() #Applies drag

            i.update\_position()

            if i.ball\_rect.bottom>=bottomline: #Collision Detection

                i.ball\_rect.bottom = bottomline

                if i.v\_y>0:

                    i.v\_y = -i.v\_y\*i.e

    pygame.display.update() #To update screen by showing newly blit surfaces

    clock.tick(30) #Adds delay of 30ms

README file

Fall Float Bouncing

Fall Float Bouncing is a simple Physics Simulator that shows gravity's effect in different mediums and environments built using the Pygame library. The objective of the project is to control the balls and observe the motion on the ground or a medium.

Features

- Control multiple balls using keyboard inputs.

- Balls bounce off the ground with realistic physics.

- Option to switch between different environments, such as "ground" and "water".

- Display velocity vectors for each ball.

- Control the density of "Water" using arrow keys.

- Drag, drop, and add balls in interactive mode.

Prerequisites

- Python 3.10

- Pygame library

Installation

1. Install the required dependencies:

pip install pygame

Usage

1. Run the script: main.py

2. Follow the on-screen instructions to interact with the project.

3. Use the spacebar to make the balls bounce.

4. Press 'A' to add a new ball.

5. Press 'R' to remove the first ball.

6. Press 'C' to switch between different environments.

7. Press 'I' to switch between interactive mode and normal mode.

7. Use the arrow keys to adjust the density (only available in certain environments).

File Structure

- `main.py`: The main script to run the project.

- `loadScreen.py`: Contains functions for loading screens and displaying the loading animation.

- `vectors.py`: Provides a function to display velocity vectors for the balls.

- `graphics/`: Directory containing various graphics assets used in the project.

Project summary:

This project demonstrates bouncing balls in different environments. The user can control the movement of the balls,

make them jump, and add or remove balls. There are three environments: ground, water, and moon. Each environment

has its own gravity setting. The project uses the Pygame library for graphics.

Credits

- Project by: Raunak Mandil

- Version: 1.0