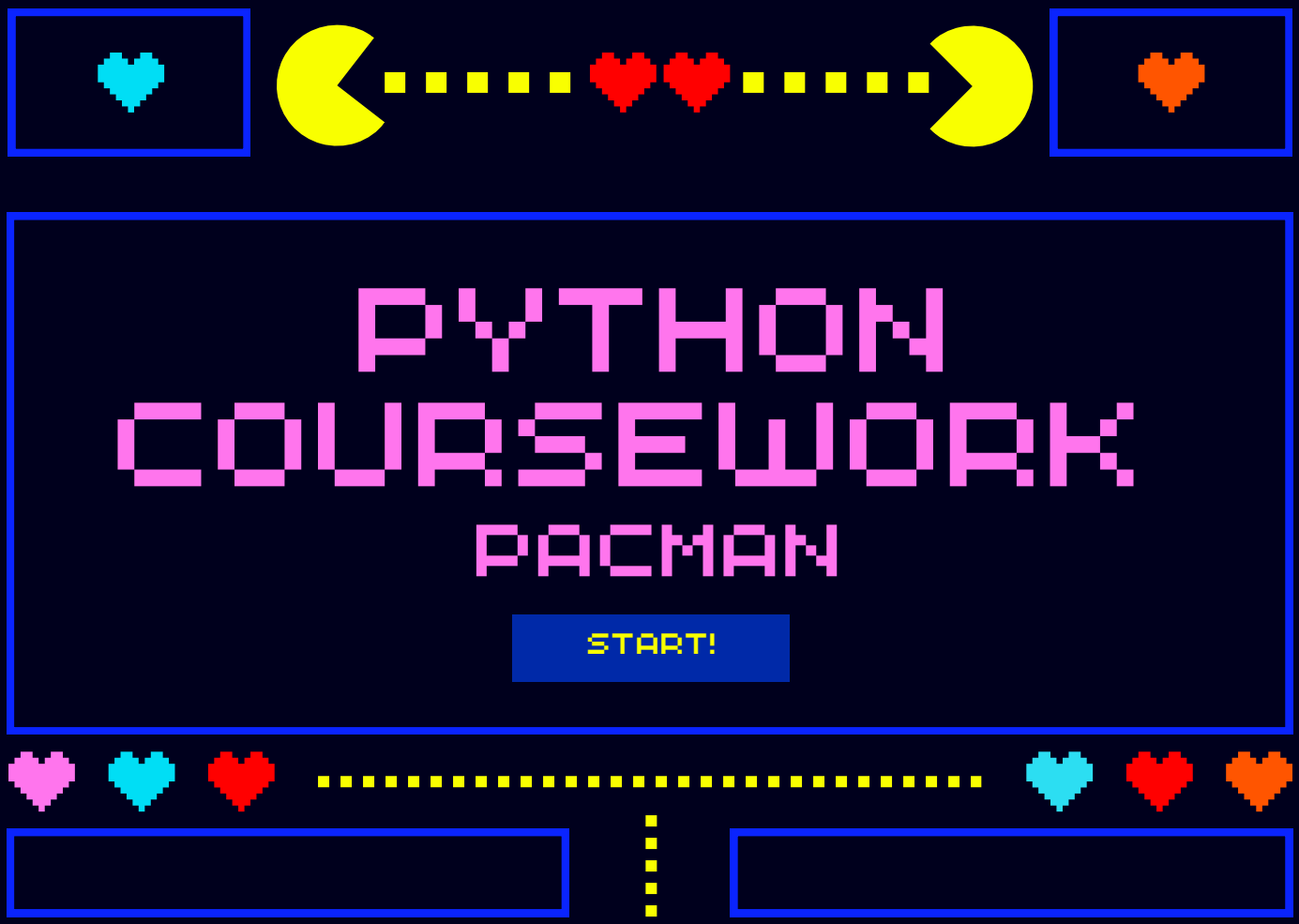


BY MAHEK TRIVEDI

CONTENTS

NEED FOR PROJECT

Introduction: Pacman, one of the most iconic arcade games of all time, continues to captivate audiences with its simple yet addictive gameplay. To bring this classic game to modern platforms and enhance its functionality, we propose the development of a Pacman game utilizing Tkinter, Pygame, threading, and OS libraries. This project aims to provide an immersive gaming experience while also exploring the integration of multiple libraries and threading techniques for enhanced performance.

Objectives:

1. Implement the core mechanics of Pacman using Pygame for graphics and event handling. Used numpy as well.
2. Utilize Tkinter to create a user-friendly graphical user interface (GUI) for menu screens and choosing your Pacman.
3. Integrate threading to keep the game loop running concurrently with the main thread, allowing for better responsiveness and handling of Pygame events
4. Incorporate OS for file operations to check if a file exists before loading data from it and to save data to a file. (Used for high score tracking)
5. Implement other minute details which were given as the project prerequisites.

USER MANUAL

1. Python 2.7.9 or higher (Distributions of Python with pip preinstalled
2. Ensure that you have Pygame, threading, numpy, tkinter and os modules installed on your device.
3. CPU: An Intel Core i5 should be your minimum, although an i7 or 8th Gen model will serve you best
4. RAM: 4 GB Minimum
5. Disk Space: 100 MB free disk space

1. Controls: Arrow keys (↑, ↓, ←, →) to move Pacman. Spacebar to pause/unpause.
2. Gameplay: Navigate Pacman, eat all pellets to advance. Avoid ghosts to avoid losing a life. Power pellets let you eat ghosts temporarily Collect fruits for extra points.
3. Scoring:
   Pellets: 10 points each.
   Ghosts and Fruits: Variable points.
4. Features: Multiple levels. High score tracking. Pause/resume. Classic Pacman gameplay.

OVERVIEW

StartMenu.py
Python program with buttons for starting, exiting, resetting score and informational dialog.

Run.py
The code manages game elements such as Pacman, ghosts, pellets, and fruit, with features including scoring, level progression, and user interaction.

Choose.py
Python program to choose if the user wants to play with Mr.Pacman, Mrs.Pacman or Baby Pacman.

Sprites.py
This code has several classes for managing sprites in a game developed using Pygame

Entity.py
Python program with methods for movement, collision detection, rendering, and attribute management.

Ghosts.py
This code defines classes for Pacman ghosts with specific behaviors, including scatter and chase modes, and manages them as a group for game development using Pygame.

Text.py
This codemanages dynamic rendering for text and updating of text content on the screen.

OVERVIEW

Pauser.py

Manages pausing and unpausing of game events or systems.
It tracks the elapsed time during a pause and can optionally execute a function after the pause ends.

Vector.py

Manages vector operations such as addition, subtraction, multiplication division, negation and checking for equal vectors using the math module.

Pacman.py

This code defines a Pacman class inheriting from the Entity class, managing Pacman's movement, collision detection, and interaction with pellets and ghosts for game development using Pygame.

Modes.py

It managea different modes of ghosts movement-scatter and chase modes, freight and spawn mode, based on a timer.

Fruit.py

The Fruit class represents a fruit object in the game that appears for a limited time (5 seconds by default). It inherits properties and behaviors from the Entity class and has its own specific attributes like color, points awarded, and a lifespan timer.

Sprites.py

This code has several classes for managing sprites in a game developed using Pygame

OVERVIEW

Animation.py

This class manages the animation of a sprite sheet. which is a collection of images that represent different frames of an animation.
It handles frame switching. speed control. looping behavior. and provides information about the current frame.

Constant.py

Constants Related to Game Size and Display . Constants for Color Definition. Constants Related to Game Elements . Constants Related to Ghost Behavior. Additional Constant.
These constants provide a clear and organized way to define various aspects of the game world. making the code more readable and maintainable

Mazedata.py

MazeBase Class:
This class defines a base structure for representing mazes in the game. It provides common functionality for managing various maze layouts.

Pellet.py

The code defines classes for representing and managing pellets and power pellets in a Pac-Man-like game.
Power pellets have a unique flashing behavior controlled by an internal timer.
The PelletGroup class provides methods for loading pellets from a data file. updating power pellet visuals. and checking if all pellets are gone.

Nodes.py

Nodes represent points of movement and decision-making within the maze.
NodeGroup manages the overall maze structure and provides access to nodes.
Access permissions for different entities (Pacman. ghosts. fruits) can be controlled at each node.

CODE

STARTMENU.PY

*StartMenu.py - C:/Users/Admin/AppData/Local/Programs/Python/Python311/Pacman_Complete/StartMenu

File Edit Format Run Options Window Help

```
from tkinter import *
from tkinter import messagebox
from PIL import Image, ImageTk
import subprocess
import random
import pygame

SCORE_FILE = "score.txt"

def update_image(count):
    frame = frames[count]
    img = ImageTk.PhotoImage(frame.resize((450, 550)))
    label.config(image=img)
    label.image = img
    count += 1
    if count == len(frames):
        count = 0
    root.after(50, update_image, count)

def start():
    stop_background_audio()
    subprocess.Popen(["python", "run.py"])
    root.destroy()

def about():
    messagebox.showinfo("Pac-Man: Classic Arcade Redux", '''
Choose Your Avatar: Select from Mr. Pac-Man, Mrs. Pac-Man, or
Baby Pac-Man for a personalized gaming experience.

Three Challenging Levels: Navigate through increasingly difficult
mazes, each with its own set of obstacles and ghostly adversaries.

Three Lives: You have three chances to conquer the maze
and achieve the highest score possible.

Strategic Gameplay: Plan your moves carefully to munch on
pellets and avoid ghosts, whose cunning increases with each level.

Timeless Fun: Experience the addictive gameplay of Pac-Man reimagined
for a new generation, offering endless hours of entertainment.''')

def exit_program():
    stop_background_audio()
    root.destroy()

def reset_score():
    with open(SCORE_FILE, "w") as file:
        file.write("0")
    messagebox.showinfo("Score Reset", "Score has been reset to 0.")
```

```
def color_change():
    new_color = "#%02x%02x%02x" % (random.randint(200, 255), random.randint(200, 255), random.randint(0, 100))
    p_button.config(fg=new_color)

def stop_background_audio():
    pygame.mixer.music.stop()

pygame.mixer.init()
pygame.mixer.music.load("pacman_beginning.wav")
pygame.mixer.music.play(loops=-1)

root = Tk()
root.geometry("500x600")
root.config(background="#000000")

gifImage = "GnS.gif"
gif = Image.open(gifImage)
frames = []
try:
    while True:
        frames.append(gif.copy().convert('RGBA'))
        gif.seek(len(frames))
except EOFError:
    pass

label = Label(root,bg="#000000")
label.place(x=0, y=0, width=500, height=600)

font_style = ('Goudy Stout', 15)
p_button = Button(root, text="PACMAN", bg="black", fg="yellow", command=color_change, font=('Goudy Stout', 18), borderwidth=0)
p_button.place(x=150, y=20)

start_button = Button(root, text="Start", command=start, font=font_style, bg="white")
start_button.place(x=169, y=200)

about_button = Button(root, text="About", command=about, font=font_style)
about_button.place(x=169, y=270)

exit_button = Button(root, text="Exit", command=exit_program, font=font_style)
exit_button.place(x=185, y=340)

reset_button = Button(root, text="Reset Score To 0?", command=reset_score, font=font_style)
reset_button.place(x=58, y=410)

update_image(0)

root.mainloop()
```

CHOOSE.PY

```python
import tkinter as tk
from PIL import Image, ImageTk

def select_avatar(avatar):
    global sp
    if avatar == "Mr Pacman":
        sp = "spritesheet.png"
    elif avatar == "Mrs Pacman":
        sp = "spritesheet_mspacman.png"
    elif avatar == "Baby Pacman":
        sp = "spritesheet_b.png"
    print("You selected:", avatar)
def close_window():
    root.destroy()

root = tk.Tk()
root.title("Choose Your Pacman")
root.configure(bg="black")
sp = ""
# Function to create a button with a resized image
def create_button_with_resized_image(image_path, avatar):
    image = Image.open(image_path)
    image = image.resize((100, 100), Image.LANCZOS)
    photo = ImageTk.PhotoImage(image)
    button = tk.Button(root, image=photo, command=lambda av=avatar: select_avatar(av))
    button.image = photo
    return button

# Images
images = [
    ("pacmanmr.png", "Mr Pacman"),
    ("mrspacman.png", "Mrs Pacman"),
    ("bpacman.png", "Baby Pacman")
]
# Create buttons with resized images
buttons = []
for img_path, avatar in images:
    button = create_button_with_resized_image(img_path, avatar)
    buttons.append(button)

# Display buttons
for i, button in enumerate(buttons):
    button.grid(row=0, column=i, padx=10, pady=10)

# Add OK button
ok_button = tk.Button(root, text="OK", command=close_window)
ok_button.grid(row=1, columnspan=len(buttons), pady=10)

root.mainloop()
```

MAIN.PY

```python
import pygame  # Importing the pygame library, which is used for creating video games.
from pygame.locals import *  # Importing constants and functions from the pygame.locals module.
from constants import *  # Importing constants from a custom module named constants.
from pacman import Pacman  # Importing the Pacman class from a custom module named pacman.
from nodes import NodeGroup  # Importing the NodeGroup class from a custom module named nodes.
from pellets import PelletGroup  # Importing the PelletGroup class from a custom module named pellets.
from ghosts import GhostGroup  # Importing the GhostGroup class from a custom module named ghosts.
from fruit import Fruit  # Importing the Fruit class from a custom module named fruit.
from pauser import Pause  # Importing the Pause class from a custom module named pauser.
from text import TextGroup  # Importing the TextGroup class from a custom module named text.
from sprites import LifeSprites, MazeSprites  # Importing the LifeSprites and MazeSprites classes from a custom module named sprites.
from mazedata import MazeData  # Importing the MazeData class from a custom module named mazedata.
import os
import threading

SCORE_FILE = "score.txt"
def game_loop(game):
    while True:
        game.update()
class GameController(object):  # Defining a class named GameController.
    def __init__(self):  # Constructor method for the GameController class.
        pygame.init()  # Initializing the pygame module.
        pygame.mixer.init()  # Initialize the mixer module for sound playback
        self.sound_chomp = pygame.mixer.Sound("pacman_chomp.wav")
        self.sound_death = pygame.mixer.Sound("pacman_death.wav")
        self.sound_intermission = pygame.mixer.Sound("pacman_intermission.wav")
        self.screen = pygame.display.set_mode(SCREENSIZE, 0, 32)  # Creating a pygame window with dimensions 448x512 pixels.
        self.background = None  # Initializing background attribute as None.
        self.background_norm = None  # Initializing background_norm attribute as None.
        self.background_flash = None  # Initializing background_flash attribute as None.
        self.clock = pygame.time.Clock()  # Creating a Clock object to help track time.
        self.fruit = None  # Initializing fruit attribute as None.
        self.pause = Pause(True)  # Creating a Pause object with initial value True.
        self.level = 0  # Initializing level attribute with value 0.
        self.lives = 3  # Initializing lives attribute with value 5.
        self.score = 0  # Initializing score attribute with value 0.
        self.textgroup = TextGroup()  # Creating a TextGroup object.
        self.lifesprites = LifeSprites(self.lives)  # Creating a LifeSprites object with initial number of lives.
        self.flashBG = False  # Initializing flashBG attribute with value False.
        self.flashTime = 0.2  # Initializing flashTime attribute with value 0.2.
        self.flashTimer = 0  # Initializing flashTimer attribute with value 0.
        self.fruitCaptured = []  # Initializing fruitCaptured attribute as an empty list.
        self.fruitNode = None  # Initializing fruitNode attribute as None.
        self.mazedata = MazeData()  # Creating a MazeData object.
        self.score = self.loadScore()  # Load the previous score from file
```

```python
    def loadScore(self):
        # Load previous score from the file
        if os.path.exists(SCORE_FILE):
            with open(SCORE_FILE, 'r') as file:
                score = int(file.read())
            return score
        else:
            return 0
    def saveScore(self):
        # Save the current score to the file
        with open(SCORE_FILE, 'w') as file:
            file.write(str(self.score))

    def setBackground(self):  # Method to set the game background.
        self.background_norm = pygame.surface.Surface(SCREENSIZE).convert()  # Creating a surface for the normal background.
        self.background_norm.fill(BLACK)  # Filling the normal background surface with black color.
        self.background_flash = pygame.surface.Surface(SCREENSIZE).convert()  # Creating a surface for the flashing background.
        self.background_flash.fill(BLACK)  # Filling the flashing background surface with black color.
        self.background_norm = self.mazesprites.constructBackground(self.background_norm, self.level % 5)  # Constructing the normal background using maze sprites.
        self.background_flash = self.mazesprites.constructBackground(self.background_flash, 5)  # Constructing the flashing background using maze sprites.
        self.flashBG = False  # Resetting flashBG attribute to False.
        self.background = self.background_norm  # Setting the background attribute to the normal background.

    def startGame(self):  # Method to start the game.
        self.mazedata.loadMaze(self.level)  # Loading maze data for the current level.
        self.mazesprites = MazeSprites(self.mazedata.obj.name + ".txt", self.mazedata.obj.name + "_rotation.txt")  # Creating MazeSprites object based on maze data.
        self.setBackground()  # Setting the game background.
        self.nodes = NodeGroup(self.mazedata.obj.name + ".txt")  # Creating NodeGroup object based on maze data.
        self.mazedata.obj.setPortalPairs(self.nodes)  # Setting portal pairs for maze nodes.
        self.mazedata.obj.connectHomeNodes(self.nodes)  # Connecting home nodes for maze nodes.
        self.pacman = Pacman(self.nodes.getNodeFromTiles(*self.mazedata.obj.pacmanStart))  # Creating Pacman object with starting node.
        self.pellets = PelletGroup(self.mazedata.obj.name + ".txt")  # Creating PelletGroup object based on maze data.
        self.ghosts = GhostGroup(self.nodes.getStartTempNode(), self.pacman)  # Creating GhostGroup object with starting node and Pacman.

        # Setting initial nodes for each ghost.
        self.ghosts.pinky.setStartNode(self.nodes.getNodeFromTiles(*self.mazedata.obj.addOffset(2, 3)))
        self.ghosts.inky.setStartNode(self.nodes.getNodeFromTiles(*self.mazedata.obj.addOffset(0, 3)))
        self.ghosts.clyde.setStartNode(self.nodes.getNodeFromTiles(*self.mazedata.obj.addOffset(4, 3)))
        self.ghosts.setSpawnNode(self.nodes.getNodeFromTiles(*self.mazedata.obj.addOffset(2, 3)))
        self.ghosts.blinky.setStartNode(self.nodes.getNodeFromTiles(*self.mazedata.obj.addOffset(2, 0)))

        # Adjust ghost speeds based on the current level
        ghost_speeds = {1: 100, 2: 110, 3: 120, 4: 130, 5: 140}  # Define speed values for each level
        ghost_speed = ghost_speeds.get(self.level, 100)  # Get the speed for the current level, default to 100 if not found
        for ghost in self.ghosts:
            ghost.setSpeed(ghost_speed)  # Set the speed for each ghost
```

```
        self.nodes.denyHomeAccess(self.pacman)  # Denying home access for Pacman.
        self.nodes.denyHomeAccessList(self.ghosts)  # Denying home access for ghosts.
        self.ghosts.inky.startNode.denyAccess(RIGHT, self.ghosts.inky)  # Denying access to right for Inky.
        self.ghosts.clyde.startNode.denyAccess(LEFT, self.ghosts.clyde)  # Denying access to left for Clyde.
        self.mazedata.obj.denyGhostsAccess(self.ghosts, self.nodes)  # Denying access for ghosts based on maze data.


    def update(self):
        dt = self.clock.tick(30) / 1000.0  # Calculate time passed since the last frame, in seconds.
        self.textgroup.update(dt)  # Update text group based on the elapsed time.
        self.pellets.update(dt)  # Update pellet group based on the elapsed time.

        # Update ghosts, fruit, and check events only if the game is not paused.
        if not self.pause.paused:
            self.ghosts.update(dt)  # Update ghosts based on the elapsed time.
            if self.fruit is not None:  # If there's a fruit in the game:
                self.fruit.update(dt)  # Update the fruit based on the elapsed time.
            self.checkPelletEvents()  # Check events related to pellets.
            self.checkGhostEvents()  # Check events related to ghosts.
            self.checkFruitEvents()  # Check events related to fruit.

        # Update Pacman's movement if it's alive and the game is not paused.
        if self.pacman.alive:
            if not self.pause.paused:
                self.pacman.update(dt)  # Update Pacman's movement based on the elapsed time.
        else:
            self.pacman.update(dt)  # Update Pacman's movement even if it's not alive.
        self.flashBG=False
        # Flash the background if needed.
        if self.flashBG:
            self.flashTimer += dt  # Increment flash timer.
            if self.flashTimer >= self.flashTime:  # If flash timer exceeds the flash time:
                self.flashTimer = 0  # Reset flash timer.
                # Toggle between normal and flashing background.
                if self.background == self.background_norm:
                    self.background = self.background_flash
                else:
                    self.background = self.background_norm

        # Update the pause status and execute any method scheduled after a pause.
        afterPauseMethod = self.pause.update(dt)
        if afterPauseMethod is not None:
            afterPauseMethod()

        self.checkEvents()  # Check for events such as quitting the game or pausing.
        self.render()  # Render all game elements onto the screen.
```

```
    def checkEvents(self):
        for event in pygame.event.get():  # Iterate through all pygame events.
            if event.type == QUIT:  # If the event is quitting the game
                game.saveScore()  #save score before exiting
                exit()  # Exit the game.
            elif event.type == KEYDOWN:  # If a key is pressed:
                if event.key == K_SPACE:  # If the pressed key is the spacebar:
                    if self.pacman.alive:  # If Pacman is alive:
                        self.pause.setPause(playerPaused=True)  # Set the game to pause mode with player pause.
                        if not self.pause.paused:  # If the game is not paused:
                            self.textgroup.hideText()  # Hide text elements.
                            self.showEntities()  # Show game entities (Pacman, ghosts, etc.).
                        else:
                            self.textgroup.showText(PAUSETXT)  # Show pause text.
                            # self.hideEntities()  # Potentially hiding entities (currently commented out).


    def checkPelletEvents(self):
        pellet = self.pacman.eatPellets(self.pellets.pelletList)  # Check if Pacman eats any pellets.
        if pellet:
            self.sound_chomp.play()# If Pacman eats a pellet:
            self.pellets.numEaten += 1  # Increase the count of pellets eaten.
            self.updateScore(pellet.points)  # Update the game score based on the pellet's points.
            if self.pellets.numEaten == 30:  # If Pacman has eaten 30 pellets:
                self.ghosts.inky.startNode.allowAccess(RIGHT, self.ghosts.inky)  # Allow Inky to move right.
            if self.pellets.numEaten == 70:  # If Pacman has eaten 70 pellets:
                self.ghosts.clyde.startNode.allowAccess(LEFT, self.ghosts.clyde)  # Allow Clyde to move left.
            if pellet in self.pellets.pelletList:
                self.pellets.pelletList.remove(pellet)  # Remove the eaten pellet from the list.
            if pellet.name == POWERPELLET:# If the eaten pellet is a power pellet:
                self.ghosts.startFreight()
                self.sound_chomp.stop()
                self.sound_intermission.play()
                # Start the freight mode for ghosts.
            if self.pellets.isEmpty():  # If there are no more pellets left:
                self.flashBG = True  # Set the background to flash.
                self.hideEntities()  # Hide game entities.
                self.pause.setPause(pauseTime=3, func=self.nextLevel)  # Pause the game and proceed to the next level after 3 seconds.

    def checkGhostEvents(self):
        for ghost in self.ghosts:  # Iterate through all ghosts.
            if self.pacman.collideGhost(ghost):  # Check if Pacman collides with the current ghost.
                if ghost.mode.current is FREIGHT:  # If the ghost is in freight mode:
                    self.pacman.visible = False  # Make Pacman invisible.
                    ghost.visible = False  # Make the ghost invisible.
                    self.updateScore(ghost.points)  # Update the score by adding the ghost's points.
                    self.textgroup.addText(str(ghost.points), WHITE, ghost.position.x, ghost.position.y, 8, time=1)  # Add text showing the points gained from eating the ghost.
                    self.ghosts.updatePoints()  # Update the points earned from eating ghosts.
                    self.pause.setPause(pauseTime=1, func=self.showEntities)  # Pause the game briefly and then show game entities.
                    ghost.startSpawn()  # Start the process of respawning the ghost.
```

```
                self.nodes.allowHomeAccess(ghost)  # Allow the ghost to access its home node.
            elif ghost.mode.current is not SPAWN:  # If the ghost is not in spawn mode:
                if self.pacman.alive:  # If Pacman is alive:
                    self.lives -= 1  # Decrease the number of lives.
                    self.lifesprites.removeImage()  # Remove one life sprite from the screen.
                    self.pacman.die()  # Mark Pacman as dead.
                    self.ghosts.hide()  # Hide all ghosts.
                    if self.lives <= 0:  # If no lives left:
                        self.textgroup.showText(GAMEOVERTXT)  # Show game over text.
                        self.pause.setPause(pauseTime=3, func=self.restartGame)  # Pause the game and restart after 3 seconds.
                    else:
                        self.sound_death.play()
                        self.pause.setPause(pauseTime=3, func=self.resetLevel)  # Pause the game and reset the level after 3 seconds.


    def checkFruitEvents(self):
        if self.pellets.numEaten == 50 or self.pellets.numEaten == 140:  # If a specific number of pellets are eaten:
            if self.fruit is None:  # If there is no existing fruit:
                self.fruit = Fruit(self.nodes.getNodeFromTiles(9, 20), self.level)  # Create a new fruit at a specific position.
                print(self.fruit)  # Print information about the created fruit.
        if self.fruit is not None:  # If there is an existing fruit:
            if self.pacman.collideCheck(self.fruit):  # If Pacman collides with the fruit:
                self.updateScore(self.fruit.points)  # Update the score by adding the fruit's points.
                self.textgroup.addText(str(self.fruit.points), WHITE, self.fruit.position.x, self.fruit.position.y, 8, time=1)  # Add text showing the points
                fruitCaptured = False  # Initialize a variable to track if the fruit is captured.
                for fruit in self.fruitCaptured:  # Iterate through captured fruits.
                    if fruit.get_offset() == self.fruit.image.get_offset():  # If the current fruit is already captured:
                        fruitCaptured = True  # Mark the fruit as captured.
                        break  # Exit the loop.
                if not fruitCaptured:  # If the fruit is not captured:
                    self.fruitCaptured.append(self.fruit.image)  # Add the fruit to the list of captured fruits.
                self.fruit = None  # Remove the existing fruit from the game.
            elif self.fruit.destroy:  # If the fruit is marked for destruction:
                self.fruit = None  # Remove the existing fruit from the game.


    def showEntities(self):
        self.pacman.visible = True  # Set Pacman to be visible.
        self.ghosts.show()  # Show all ghosts.


    def hideEntities(self):
        self.pacman.visible = False  # Set Pacman to be invisible.
        self.ghosts.hide()  # Hide all ghosts.


    def nextLevel(self):
        self.showEntities()  # Show game entities (Pacman and ghosts).
        self.level += 1  # Increment the level number.
        self.pause.paused = True  # Pause the game.
```

```
        self.startGame()  # Start the next level.
        self.textgroup.updateLevel(self.level)  # Update the displayed level number.
    def restartGame(self):
        self.lives = 3
        self.level = 0
        self.pause.paused = True
        self.fruit = None
        self.startGame()
        self.score = self.loadScore()  # Load previous score
        self.textgroup.updateScore(self.score)  # Update displayed score
        self.textgroup.updateLevel(self.level)
        self.textgroup.showText(READYTXT)
        self.lifesprites.resetLives(self.lives)
        self.fruitCaptured = []
    def updateScore(self, points):
        self.score += points
        self.textgroup.updateScore(self.score)
        self.saveScore()  # Save the score after each update
    def resetLevel(self):
        self.pause.paused = True  # Pause the game.
        self.pacman.reset()  # Reset Pacman's position and state.
        self.ghosts.reset()  # Reset all ghosts.
        self.fruit = None  # Clear any existing fruit.
        self.textgroup.showText(READYTXT)  # Show the "READY" text.
    def render(self):
        self.screen.blit(self.background, (0, 0))  # Render the background onto the screen.
        self.pellets.render(self.screen)  # Render pellets onto the screen.
        if self.fruit is not None:  # If there is a fruit:
            self.fruit.render(self.screen)  # Render the fruit onto the screen.
        self.pacman.render(self.screen)  # Render Pacman onto the screen.
        self.ghosts.render(self.screen)  # Render ghosts onto the screen.
        self.textgroup.render(self.screen)  # Render text elements onto the screen.
        for i in range(len(self.lifesprites.images)): # Render life sprites onto the screen.
            x = self.lifesprites.images[i].get_width() * i
            y = SCREENHEIGHT - self.lifesprites.images[i].get_height()
            self.screen.blit(self.lifesprites.images[i], (x, y))
        for i in range(len(self.fruitCaptured)):# Render captured fruit images onto the screen.
            x = SCREENWIDTH - self.fruitCaptured[i].get_width() * (i+1)
            y = SCREENHEIGHT - self.fruitCaptured[i].get_height()
            self.screen.blit(self.fruitCaptured[i], (x, y))
        pygame.display.update()  # Update the display to show all rendered elements.
if __name__ == "__main__":
    game = GameController()
    game.startGame()
    game_thread = threading.Thread(target=game_loop, args=(game,))# Start the game loop in a separate thread
    game_thread.daemon = True  # Daemonize the thread so it exits when the main thread exits
    game_thread.start()
    # Call game.update() here to keep the main loop running and handle Pygame events
    game_loop(game)
```

ENTITY.PY

```python
import pygame  # Import the pygame library for game development.
from pygame.locals import *  # Import constants from pygame.locals module.
from vector import Vector2  # Import Vector2 class from vector module.
from constants import *  # Import constants from constants module.
from random import randint  # Import randint function from random module.

class Entity(object):  # Define a class named Entity.
    def __init__(self, node):  # Constructor method for initializing instances of the Entity class.
        self.name = None  # Initialize the name attribute to None.
        # Define movement directions with corresponding vectors.
        self.directions = {UP:Vector2(0, -1), DOWN:Vector2(0, 1), LEFT:Vector2(-1, 0), RIGHT:Vector2(1, 0), STOP:Vector2()}
        self.direction = STOP  # Initialize the direction attribute to STOP.
        self.setSpeed(100)  # Set the speed attribute.
        self.radius = 10  # Radius of the entity.
        self.collideRadius = 3  # Radius for collision detection.
        self.color = WHITE  # Color of the entity.
        self.visible = True  # Boolean indicating if the entity is visible.
        self.disablePortal = False  # Boolean indicating if portals are disabled.
        self.goal = None  # Initialize the goal attribute to None.
        # Define the default method for determining direction.
        self.directionMethod = self.randomDirection
        self.setStartNode(node)  # Set the starting node for the entity.
        self.image = None  # Initialize the image attribute to None.

    def setPosition(self):  # Method to set the position of the entity.
        self.position = self.node.position.copy()  # Set the position attribute.

    def update(self, dt):  # Method to update the entity's state.
        # Update the position based on direction and speed.
        self.position += self.directions[self.direction] * self.speed * dt
        # Check if the entity has overshot its target node.
        if self.overshotTarget():
            self.node = self.target  # Update the current node to the target node.
            directions = self.validDirections()  # Get valid movement directions.
            direction = self.directionMethod(directions)  # Determine next direction.
            if not self.disablePortal:
                # Check if the current node has a portal neighbor.
                if self.node.neighbors[PORTAL] is not None:
                    self.node = self.node.neighbors[PORTAL]  # Move through the portal.
            self.target = self.getNewTarget(direction)  # Determine the new target node.
            if self.target is not self.node:
                self.direction = direction  # Update the direction.
            else:
                self.target = self.getNewTarget(self.direction)  # Keep moving in the current direction.
            self.setPosition()  # Set the new position.

    def validDirection(self, direction):  # Method to check if a direction is valid.
        if direction is not STOP:  # Ensure direction is not STOP.
            if self.name in self.node.access[direction]:  # Check if the entity can access the direction from the current node.
                if self.node.neighbors[direction] is not None:  # Check if there is a neighbor in the specified direction.
                    return True  # Direction is valid.
        return False  # Direction is invalid.

    def getNewTarget(self, direction):  # Method to get the new target node based on direction.
        if self.validDirection(direction):  # Check if the direction is valid.
            return self.node.neighbors[direction]  # Return the neighbor node in the specified direction.
        return self.node  # Return the current node if direction is invalid.

    def overshotTarget(self):  # Method to check if the entity has overshot its target node.
        if self.target is not None:  # Ensure target node is defined.
            vec1 = self.target.position - self.node.position  # Vector from current node to target node.
            vec2 = self.position - self.node.position  # Vector from current node to current position.
            node2Target = vec1.magnitudeSquared()  # Squared magnitude of vector to target node.
            node2Self = vec2.magnitudeSquared()  # Squared magnitude of vector to current position.
            return node2Self >= node2Target  # Check if current position is beyond target node.
        return False  # Return False if target node is not defined.

    def reverseDirection(self):  # Method to reverse the entity's direction.
        self.direction *= -1  # Reverse the direction.
        temp = self.node  # Swap current node and target node.
        self.node = self.target
        self.target = temp

    def oppositeDirection(self, direction):  # Method to check if a direction is opposite to the entity's current direction.
        if direction is not STOP:  # Ensure direction is not STOP.
            if direction == self.direction * -1:  # Check if direction is opposite to current direction.
                return True  # Direction is opposite.
        return False  # Direction is not opposite.

    def validDirections(self):  # Method to get valid movement directions.
        directions = []  # List to store valid directions.
        for key in [UP, DOWN, LEFT, RIGHT]:  # Iterate over possible directions.
            if self.validDirection(key):  # Check if direction is valid.
                if key != self.direction * -1:  # Ensure direction is not opposite to current direction.
                    directions.append(key)  # Add valid direction to list.
        if len(directions) == 0:  # If no valid directions found.
            directions.append(self.direction * -1)  # Add opposite direction to list.
        return directions  # Return list of valid directions.

    def randomDirection(self, directions):  # Method to select a random direction from a list of valid directions.
        return directions[randint(0, len(directions) - 1)]  # Return a randomly selected direction.

    def goalDirection(self, directions):  # Method to select direction towards a goal.
        distances = []  # List to store distances to goal for each direction.
        for direction in directions:  # Iterate over valid directions.
            vec = self.node.position + self.directions[direction] * TILEWIDTH - self.goal  # Vector to goal.
            distances.append(vec.magnitudeSquared())  # Calculate squared distance and append to list.
        index = distances.index(min(distances))  # Find index of direction with minimum distance to goal.
        return directions[index]  # Return direction towards goal.
```

```
    def setStartNode(self, node):  # Method to set the starting node for the entity.
        self.node = node  # Set current node to starting node.
        self.startNode = node  # Set starting node attribute.
        self.target = node  # Set target node to starting node.
        self.setPosition()  # Set position to starting node position.
    def setBetweenNodes(self, direction):
        # Method to set the entity between two nodes in a specified direction.
        if self.node.neighbors[direction] is not None:
            # Check if there is a neighbor node in the specified direction.
            self.target = self.node.neighbors[direction]
            # Set the target node to the neighbor node in the specified direction.
            self.position = (self.node.position + self.target.position) / 2.0
            # Set the position of the entity to the midpoint between the current node and the target node.
    def reset(self):
        # Method to reset the entity's attributes to default values.
        self.setStartNode(self.startNode)
        # Reset the entity's starting node.
        self.direction = STOP
        # Reset the entity's direction to STOP.
        self.speed = 100
        # Reset the entity's speed to 100.
        self.visible = True
        # Set the entity's visibility to True.
    def setSpeed(self, speed):
        # Method to set the speed of the entity.
        self.speed = speed * TILEWIDTH / 16
        # Set the entity's speed, adjusting it based on TILEWIDTH.
    def render(self, screen):
        # Method to render the entity on the screen.
        if self.visible:
            # Check if the entity is visible.
            if self.image is not None:
                # If the entity has an image assigned to it.
                adjust = Vector2(TILEWIDTH, TILEHEIGHT) / 2
                # Calculate adjustment based on TILEWIDTH and TILEHEIGHT.
                p = self.position - adjust
                # Calculate the position of the entity.
                screen.blit(self.image, p.asTuple())
                # Draw the image of the entity on the screen at the calculated position.
            else:
                p = self.position.asInt()
                # If no image is assigned, draw a circle representing the entity.
                pygame.draw.circle(screen, self.color, p, self.radius)
                # Draw a circle representing the entity on the screen.
```

GHOSTS.PY

```python
import pygame  # Import the pygame library for game development
from pygame.locals import *  # Import specific constants from pygame.locals module
from vector import Vector2  # Importing a custom Vector2 class from a module named vector
from constants import *  # Importing constants from a module named constants
from entity import Entity  # Importing Entity class from a module named entity
from modes import ModeController  # Importing ModeController class from a module named modes
from sprites import GhostSprites  # Importing GhostSprites class from a module named sprites
# Define a class Ghost which inherits from Entity
class Ghost(Entity):
    # Constructor method to initialize Ghost objects
    def __init__(self, node, pacman=None, blinky=None):
        # Call the constructor of the superclass (Entity)
        Entity.__init__(self, node)
        # Set attributes specific to Ghost objects
        self.name = GHOST  # Name of the ghost
        self.points = 200  # Points earned when ghost is captured
        self.goal = Vector2()  # Target position for the ghost
        self.directionMethod = self.goalDirection  # Method for determining ghost's movement direction
        self.pacman = pacman  # Reference to the Pacman object
        self.mode = ModeController(self)  # Mode controller for ghost's behavior
        self.blinky = blinky  # Reference to the Blinky ghost object
        self.homeNode = node  # Node where ghost starts
    # Method to reset ghost's attributes
    def reset(self):
        Entity.reset(self)  # Call reset method of superclass (Entity)
        self.points = 200  # Reset points earned
        self.directionMethod = self.goalDirection  # Reset method for determining movement direction
    # Method to update ghost's state
    def update(self, dt):
        self.sprites.update(dt)  # Update ghost's sprites
        self.mode.update(dt)  # Update ghost's mode (scatter, chase, etc.)
        if self.mode.current is SCATTER:  # If current mode is scatter
            self.scatter()  # Execute scatter behavior
        elif self.mode.current is CHASE:  # If current mode is chase
            self.chase()  # Execute chase behavior
        Entity.update(self, dt)  # Call update method of superclass (Entity)
    # Method defining ghost's behavior when in scatter mode
    def scatter(self):
        self.goal = Vector2()  # Set goal position to default (no specific target)
    # Method defining ghost's behavior when in chase mode
    def chase(self):
        self.goal = self.pacman.position  # Set goal position to Pacman's current position
    # Method to spawn the ghost
    def spawn(self):
        self.goal = self.spawnNode.position  # Set goal position to spawn node's position
    # Method to set the node where the ghost spawns
    def setSpawnNode(self, node):
        self.spawnNode = node  # Set spawn node attribute to given node
```

```python
    # Method to start ghost's spawn mode
    def startSpawn(self):
        self.mode.setSpawnMode()  # Set ghost's mode to spawn
        if self.mode.current == SPAWN:  # If current mode is spawn
            self.setSpeed(150)  # Set ghost's speed
            self.directionMethod = self.goalDirection  # Set direction method
            self.spawn()  # Spawn the ghost

    # Method to start ghost's freight mode
    def startFreight(self):
        self.mode.setFreightMode()  # Set ghost's mode to freight
        if self.mode.current == FREIGHT:  # If current mode is freight
            self.setSpeed(50)  # Set ghost's speed
            self.directionMethod = self.randomDirection  # Set direction method

    # Method to set ghost's behavior in normal mode
    def normalMode(self):
        self.setSpeed(100)  # Set ghost's speed
        self.directionMethod = self.goalDirection  # Set direction method
        self.homeNode.denyAccess(DOWN, self)  # Deny access to a particular direction at home node

# Define a subclass Blinky which inherits from Ghost
class Blinky(Ghost):
    # Constructor method to initialize Blinky objects
    def __init__(self, node, pacman=None, blinky=None):
        # Call the constructor of the superclass (Ghost)
        Ghost.__init__(self, node, pacman, blinky)
        # Set attributes specific to Blinky objects
        self.name = BLINKY  # Name of the ghost
        self.color = RED  # Color of the ghost
        self.sprites = GhostSprites(self)  # Initialize ghost sprites

# Define a subclass Pinky which inherits from Ghost
class Pinky(Ghost):
    # Constructor method to initialize Pinky objects
    def __init__(self, node, pacman=None, blinky=None):
        # Call the constructor of the superclass (Ghost)
        Ghost.__init__(self, node, pacman, blinky)
        # Set attributes specific to Pinky objects
        self.name = PINKY  # Name of the ghost
        self.color = PINK  # Color of the ghost
        self.sprites = GhostSprites(self)  # Initialize ghost sprites

    # Override scatter method for Pinky's specific behavior
    def scatter(self):
        self.goal = Vector2(TILEWIDTH*NCOLS, 0)  # Set goal to top-right corner of the maze

    # Override chase method for Pinky's specific behavior
    def chase(self):
        self.goal = self.pacman.position + self.pacman.directions[self.pacman.direction] * TILEWIDTH * 4  # Set goal to position four tiles ahead of Pacman
```

```python
# Define a subclass Inky which inherits from Ghost
class Inky(Ghost):
    # Constructor method to initialize Inky objects
    def __init__(self, node, pacman=None, blinky=None):
        # Call the constructor of the superclass (Ghost)
        Ghost.__init__(self, node, pacman, blinky)
        # Set attributes specific to Inky objects
        self.name = INKY  # Name of the ghost
        self.color = TEAL  # Color of the ghost
        self.sprites = GhostSprites(self)  # Initialize ghost sprites

    # Override scatter method for Inky's specific behavior
    def scatter(self):
        self.goal = Vector2(TILEWIDTH*NCOLS, TILEHEIGHT*NROWS)  # Set goal to bottom-right corner of the maze

    # Override chase method for Inky's specific behavior
    def chase(self):
        vec1 = self.pacman.position + self.pacman.directions[self.pacman.direction] * TILEWIDTH * 2  # Calculate a vector two tiles ahead of Pacman
        vec2 = (vec1 - self.blinky.position) * 2  # Calculate a vector from Blinky to the calculated point, then double its length
        self.goal = self.blinky.position + vec2  # Set goal to Blinky's position plus the calculated vector

class Clyde(Ghost):
    def __init__(self, node, pacman=None, blinky=None):
        # Initialize the Clyde ghost
        Ghost.__init__(self, node, pacman, blinky)  # Call the __init__ method of the Ghost class
        self.name = CLYDE  # Set the name of the ghost to "CLYDE"
        self.color = ORANGE  # Set the color of the ghost to ORANGE
        self.sprites = GhostSprites(self)  # Initialize sprites for Clyde ghost

    def scatter(self):
        # Set the goal for Clyde ghost during scatter mode
        self.goal = Vector2(0, TILEHEIGHT*NROWS)  # Set the goal position to the bottom-left corner of the maze

    def chase(self):
        # Set the goal for Clyde ghost during chase mode
        d = self.pacman.position - self.position  # Calculate the vector from Clyde to Pacman
        ds = d.magnitudeSquared()  # Calculate the squared magnitude of the vector d
        if ds <= (TILEWIDTH * 8)**2:  # If Pacman is within a certain range of Clyde
            self.scatter()  # Set the goal to scatter mode
        else:
            # Otherwise, set the goal to follow Pacman but at a distance ahead
            self.goal = self.pacman.position + self.pacman.directions[self.pacman.direction] * TILEWIDTH * 4
```

```python
class GhostGroup(object):
    def __init__(self, node, pacman):
        # Initialize a group of ghosts
        self.blinky = Blinky(node, pacman)  # Initialize Blinky ghost
        self.pinky = Pinky(node, pacman)  # Initialize Pinky ghost
        self.inky = Inky(node, pacman, self.blinky)  # Initialize Inky ghost
        self.clyde = Clyde(node, pacman)  # Initialize Clyde ghost
        self.ghosts = [self.blinky, self.pinky, self.inky, self.clyde]  # Store all ghosts in a list
    def __iter__(self):
        return iter(self.ghosts)  # Allow iteration over the ghosts in the group
    def update(self, dt):
        # Update all ghosts in the group
        for ghost in self:
            ghost.update(dt)
    def startFreight(self):
        # Start freight mode for all ghosts in the group
        for ghost in self:
            ghost.startFreight()
        self.resetPoints()  # Reset points for all ghosts
    def setSpawnNode(self, node):
        # Set spawn node for all ghosts in the group
        for ghost in self:
            ghost.setSpawnNode(node)
    def updatePoints(self):
        # Update points for all ghosts in the group
        for ghost in self:
            ghost.points *= 2
    def resetPoints(self):
        # Reset points for all ghosts in the group
        for ghost in self:
            ghost.points = 200
    def hide(self):
        # Hide all ghosts in the group
        for ghost in self:
            ghost.visible = False
    def show(self):
        # Show all ghosts in the group
        for ghost in self:
            ghost.visible = True
    def reset(self):
        # Reset all ghosts in the group
        for ghost in self:
            ghost.reset()
    def render(self, screen):
        # Render all ghosts in the group on the screen
        for ghost in self:
            ghost.render(screen)
```

MODES.PY

```python
# Import constants from the constants FILE
from constants import *
# Define a class MainMode
class MainMode(object):
    # Constructor method
    def __init__(self):
        # Initialize timer variable to 0
        self.timer = 0
        # Call the scatter method upon initialization
        self.scatter()
    # Update method to update the mode
    def update(self, dt):
        # Increment timer by the time difference (dt)
        self.timer += dt
        # Check if timer exceeds the set time for the mode
        if self.timer >= self.time:
            # If mode is SCATTER, switch to chase mode
            if self.mode is SCATTER:
                self.chase()
            # If mode is CHASE, switch to scatter mode
            elif self.mode is CHASE:
                self.scatter()
    # Method to set the mode to scatter
    def scatter(self):
        # Set mode to SCATTER
        self.mode = SCATTER
        # Set time for this mode
        self.time = 7
        # Reset timer
        self.timer = 0
    # Method to set the mode to chase
    def chase(self):
        # Set mode to CHASE
        self.mode = CHASE
        # Set time for this mode
        self.time = 20
        # Reset timer
        self.timer = 0
# Define a class ModeController
class ModeController(object):
    # Constructor method
    def __init__(self, entity):
        # Initialize timer variable to 0
        self.timer = 0
        # Set time to None
        self.time = None
        # Create an instance of MainMode class
        self.mainmode = MainMode()
        # Set current mode to the mode of MainMode instance
        self.current = self.mainmode.mode
```

```
        # Assign the entity to the controller
        self.entity = entity
    # Method to update the mode
    def update(self, dt):
        # Call update method of MainMode instance
        self.mainmode.update(dt)
        # Check if current mode is FREIGHT
        if self.current is FREIGHT:
            # Increment timer by time difference (dt)
            self.timer += dt
            # Check if timer exceeds set time
            if self.timer >= self.time:
                # Set time to None
                self.time = None
                # Call entity's normal mode method
                self.entity.normalMode()
                # Set current mode to the mode of MainMode instance
                self.current = self.mainmode.mode
        elif self.current in [SCATTER, CHASE]: # Check if current mode is SCATTER or CHASE
            # Set current mode to the mode of MainMode instance
            self.current = self.mainmode.mode
        # Check if current mode is SPAWN
        if self.current is SPAWN:
            # Check if entity is at spawn node
            if self.entity.node == self.entity.spawnNode:
                # Call entity's normal mode method
                self.entity.normalMode()
                # Set current mode to the mode of MainMode instance
                self.current = self.mainmode.mode
    # Method to set the mode to FREIGHT
    def setFreightMode(self):
        # Check if current mode is SCATTER or CHASE
        if self.current in [SCATTER, CHASE]:
            # Reset timer
            self.timer = 0
            # Set time for FREIGHT mode
            self.time = 7
            # Set current mode to FREIGHT
            self.current = FREIGHT
        # Check if current mode is already FREIGHT
        elif self.current is FREIGHT:
            # Reset timer
            self.timer = 0
    # Method to set the mode to SPAWN
    def setSpawnMode(self):
        # Check if current mode is FREIGHT
        if self.current is FREIGHT:
            # Set current mode to SPAWN
            self.current = SPAWN
```

PACMAN.PY

```python
import pygame  # Importing the pygame library, used for developing games in Python.
from pygame.locals import *  # Importing specific constants and functions from pygame.locals module.
from vector import Vector2  # Importing the Vector2 class from a custom module named vector.
from constants import *  # Importing constants from a custom module named constants.
from entity import Entity  # Importing the Entity class from a custom module named entity.
from sprites import PacmanSprites  # Importing the PacmanSprites class from a custom module named sprites.

class Pacman(Entity):  # Defining a class named Pacman, which inherits from the Entity class.
    def __init__(self, node):  # Constructor method for Pacman class, takes a node parameter.
        Entity.__init__(self, node)  # Calling the constructor of the parent class with the node parameter.
        self.name = PACMAN  # Setting the name attribute of Pacman object to the value of PACMAN constant.
        self.color = YELLOW  # Setting the color attribute of Pacman object to the value of YELLOW constant.
        self.direction = LEFT  # Setting the initial direction attribute of Pacman object to LEFT.
        self.setBetweenNodes(LEFT)  # Calling a method to set Pacman between nodes in the LEFT direction.
        self.alive = True  # Setting the alive attribute of Pacman object to True.
        self.sprites = PacmanSprites(self)  # Creating an instance of PacmanSprites class and assigning it to sprites attribute.

    def reset(self):  # Method to reset Pacman object to its initial state.
        Entity.reset(self)  # Calling the reset method of the parent class.
        self.direction = LEFT  # Resetting the direction attribute to LEFT.
        self.setBetweenNodes(LEFT)  # Resetting Pacman between nodes in the LEFT direction.
        self.alive = True  # Setting the alive attribute to True.
        self.image = self.sprites.getStartImage()  # Setting the image attribute using a method from the sprites attribute.
        self.sprites.reset()  # Calling the reset method of the sprites attribute.

    def die(self):  # Method to handle Pacman's death.
        self.alive = False  # Setting the alive attribute to False.
        self.direction = STOP  # Setting the direction attribute to STOP.

    def update(self, dt):  # Method to update Pacman's state.
        self.sprites.update(dt)  # Calling the update method of the sprites attribute.
        self.position += self.directions[self.direction] * self.speed * dt  # Updating Pacman's position based on direction, speed, and time.
        direction = self.getValidKey()  # Getting a valid direction from user input.
        if self.overshotTarget():  # Checking if Pacman has overshot its target node.
            # Handling movement when Pacman reaches a target node.
            self.node = self.target
            if self.node.neighbors[PORTAL] is not None:
                self.node = self.node.neighbors[PORTAL]
            self.target = self.getNewTarget(direction)
            if self.target is not self.node:
                self.direction = direction
            else:
                self.target = self.getNewTarget(self.direction)

            if self.target is self.node:
                self.direction = STOP
            self.setPosition()
        else:
            if self.oppositeDirection(direction):
                self.reverseDirection()  # Reversing Pacman's direction if it's opposite to the current direction.
```

```python
    def getValidKey(self):  # Method to get valid input direction from the user.
        key_pressed = pygame.key.get_pressed()  # Getting the state of all keyboard keys.
        if key_pressed[K_UP]:
            return UP
        if key_pressed[K_DOWN]:
            return DOWN
        if key_pressed[K_LEFT]:
            return LEFT
        if key_pressed[K_RIGHT]:
            return RIGHT
        return STOP  # Returning STOP if no valid key is pressed.

    def eatPellets(self, pelletList):  # Method to handle Pacman eating pellets.
        for pellet in pelletList:
            if self.collideCheck(pellet):  # Checking collision between Pacman and pellets.
                return pellet
        return None  # Returning None if no pellet is eaten.

    def collideGhost(self, ghost):  # Method to check collision with ghosts.
        return self.collideCheck(ghost)  # Checking collision between Pacman and ghosts.

    def collideCheck(self, other):  # Method to check collision between Pacman and other objects.
        d = self.position - other.position  # Calculating the distance between Pacman and the other object.
        dSquared = d.magnitudeSquared()  # Calculating the squared magnitude of the distance vector.
        rSquared = (self.collideRadius + other.collideRadius) ** 2  # Calculating the sum of squared radii.
        if dSquared <= rSquared:  # Checking if the squared distance is less than or equal to the squared sum of radii.
            return True  # Returning True if collision occurs.
        return False  # Returning False if no collision occurs.
```

SPRITES.PY

```
import pygame  # Importing the pygame library for game development.
from constants import *  # Importing constants module containing game constants.
import numpy as np  # Importing numpy library for numerical operations.
from animation import Animator  # Importing Animator class from animation module.
from Choose import *  # Importing Choose module.

BASETILEWIDTH = 16  # Setting the base width of a tile.
BASETILEHEIGHT = 16  # Setting the base height of a tile.
DEATH = 5  # Constant representing the death state.
numlives = 3  # Initializing the number of lives.
class Spritesheet(object):  # Defining a class for handling sprite sheets.
    def __init__(self):  # Constructor method for Spritesheet class.
        self.sheet = pygame.image.load(sp).convert()  # Loading sprite sheet image.
        transcolor = self.sheet.get_at((0,0))  # Getting the color of the top-left pixel as transparent color.
        self.sheet.set_colorkey(transcolor)  # Setting transparent color for the sprite sheet.
        width = int(self.sheet.get_width() / BASETILEWIDTH * TILEWIDTH)  # Calculating width of sprite sheet in pixels.
        height = int(self.sheet.get_height() / BASETILEHEIGHT * TILEHEIGHT)  # Calculating height of sprite sheet in pixels.
        self.sheet = pygame.transform.scale(self.sheet, (width, height))  # Scaling the sprite sheet.

    def getImage(self, x, y, width, height):  # Method to extract image from sprite sheet.
        x *= TILEWIDTH  # Calculating x-coordinate of image in pixels.
        y *= TILEHEIGHT  # Calculating y-coordinate of image in pixels.
        self.sheet.set_clip(pygame.Rect(x, y, width, height))  # Setting clipping region on sprite sheet.
        return self.sheet.subsurface(self.sheet.get_clip())  # Returning the clipped portion as image.
class PacmanSprites(Spritesheet):  # Class for handling Pacman sprites.
    def __init__(self, entity):  # Constructor method for PacmanSprites class.
        Spritesheet.__init__(self)  # Calling constructor of parent class.
        self.entity = entity  # Storing the Pacman entity.
        self.entity.image = self.getStartImage()  # Setting the initial image for Pacman.
        self.animations = {}  # Initializing dictionary to hold animations.
        self.defineAnimations()  # Defining animations for Pacman.
        self.stopimage = (8, 0)  # Default image when Pacman stops moving.
    def defineAnimations(self):  # Method to define animations for Pacman.
        # Defining animations for different directions and actions.
        self.animations[LEFT] = Animator(((8,0), (0, 0), (0, 2), (0, 0)))
        self.animations[RIGHT] = Animator(((10,0), (2, 0), (2, 2), (2, 0)))
        self.animations[UP] = Animator(((10,2), (6, 0), (6, 2), (6, 0)))
        self.animations[DOWN] = Animator(((8,2), (4, 0), (4, 2), (4, 0)))
        self.animations[DEATH] = Animator(((0, 12), (2, 12), (4, 12), (6, 12), (8, 12), (10, 12), (12, 12), (14, 12), (16, 12), (18, 12), (20, 12)), speed=6, loop=False)
    def update(self, dt):  # Method to update Pacman's animation.
        if self.entity.alive == True:  # Checking if Pacman is alive.
            if self.entity.direction == LEFT:  # If Pacman is moving left.
                self.entity.image = self.getImage(*self.animations[LEFT].update(dt))  # Updating image based on animation frame.
                self.stopimage = (8, 0)  # Setting the default stop image for left direction.
            elif self.entity.direction == RIGHT:  # If Pacman is moving right.
                self.entity.image = self.getImage(*self.animations[RIGHT].update(dt))  # Updating image based on animation frame.
                self.stopimage = (10, 0)  # Setting the default stop image for right direction.
            elif self.entity.direction == DOWN:  # If Pacman is moving down.
                self.entity.image = self.getImage(*self.animations[DOWN].update(dt))  # Updating image based on animation frame.
                self.stopimage = (8, 2)  # Setting the default stop image for down direction.
```

```
            elif self.entity.direction == UP:  # If Pacman is moving up.
                self.entity.image = self.getImage(*self.animations[UP].update(dt))  # Updating image based on animation frame.
                self.stopimage = (10, 2)  # Setting the default stop image for up direction.
            elif self.entity.direction == STOP:  # If Pacman is not moving.
                self.entity.image = self.getImage(*self.stopimage)  # Setting the stop image.
        else:  # If Pacman is not alive.
            self.entity.image = self.getImage(*self.animations[DEATH].update(dt))  # Updating image based on death animation.

    def reset(self):  # Method to reset animations.
        for key in list(self.animations.keys()):  # Iterating over animation keys.
            self.animations[key].reset()  # Resetting each animation.

    def getStartImage(self):  # Method to get the starting image for Pacman.
        return self.getImage(8, 0)  # Returning the image for Pacman facing left.

    def getImage(self, x, y):  # Method to get image from sprite sheet.
        return Spritesheet.getImage(self, x, y, 2*TILEWIDTH, 2*TILEHEIGHT)  # Returning the image.


class GhostSprites(Spritesheet):  # Class for handling Ghost sprites.
    def __init__(self, entity):  # Constructor method for GhostSprites class.
        Spritesheet.__init__(self)  # Calling constructor of parent class.
        self.x = {BLINKY:0, PINKY:2, INKY:4, CLYDE:6}  # Dictionary to map ghost names to their x-coordinate.
        self.entity = entity  # Storing the Ghost entity.
        self.entity.image = self.getStartImage()  # Setting the initial image for Ghost.

    def update(self, dt):  # Method to update Ghost's animation.
        x = self.x[self.entity.name]  # Getting the x-coordinate of the Ghost.

        # Checking the current mode of the Ghost.
        if self.entity.mode.current in [SCATTER, CHASE]:  # If Ghost is in scatter or chase mode.
            if self.entity.direction == LEFT:  # If Ghost is moving left.
                self.entity.image = self.getImage(x, 8)  # Setting image for left movement.
            elif self.entity.direction == RIGHT:  # If Ghost is moving right.
                self.entity.image = self.getImage(x, 10)  # Setting image for right movement.
            elif self.entity.direction == DOWN:  # If Ghost is moving down.
                self.entity.image = self.getImage(x, 6)  # Setting image for downward movement.
            elif self.entity.direction == UP:  # If Ghost is moving up.
                self.entity.image = self.getImage(x, 4)  # Setting image for upward movement.
        elif self.entity.mode.current == FREIGHT:  # If Ghost is in fright mode.
            self.entity.image = self.getImage(10, 4)  # Setting fright mode image.
        elif self.entity.mode.current == SPAWN:  # If Ghost is in spawn mode.
            if self.entity.direction == LEFT:  # If Ghost is moving left.
                self.entity.image = self.getImage(8, 8)  # Setting image for leftward spawn.
            elif self.entity.direction == RIGHT:  # If Ghost is moving right.
                self.entity.image = self.getImage(8, 10)  # Setting image for rightward spawn.
            elif self.entity.direction == DOWN:  # If Ghost is moving down.
                self.entity.image = self.getImage(8, 6)  # Setting image for downward spawn.
            elif self.entity.direction == UP:  # If Ghost is moving up.
                self.entity.image = self.getImage(8, 4)  # Setting image for upward spawn.
```

```
    def getStartImage(self):  # Method to get the starting image for Ghost.
        return self.getImage(self.x[self.entity.name], 4)  # Returning the image for Ghost's starting position.

    def getImage(self, x, y):  # Method to get image from sprite sheet.
        return Spritesheet.getImage(self, x, y, 2*TILEWIDTH, 2*TILEHEIGHT)  # Returning the image.



class FruitSprites(Spritesheet):
    # Define a class named FruitSprites that inherits from the Spritesheet class
    def __init__(self, entity, level):
        # Constructor method to initialize an instance of the FruitSprites class
        Spritesheet.__init__(self)
        # Call the constructor of the parent class (Spritesheet) to initialize inherited attributes
        self.entity = entity
        # Assign the entity parameter to the instance attribute 'entity'
        self.fruits = {0:(16,8), 1:(18,8), 2:(20,8), 3:(16,10), 4:(18,10), 5:(20,10)}
        # Define a dictionary 'fruits' with keys representing fruit types and values representing their coordinates
        self.entity.image = self.getStartImage(level % len(self.fruits))
        # Set the image attribute of the entity to the result of calling the getStartImage method with a computed index based on the level parameter

    def getStartImage(self, key):
        # Define a method named getStartImage which takes a key as input
        return self.getImage(*self.fruits[key])
        # Call the getImage method with coordinates corresponding to the key in the fruits dictionary and return the result

    def getImage(self, x, y):
        # Define a method named getImage which takes x and y coordinates as input
        return Spritesheet.getImage(self, x, y, 2*TILEWIDTH, 2*TILEHEIGHT)
        # Call the getImage method of the parent class (Spritesheet) with adjusted coordinates and return the result


class LifeSprites(Spritesheet):
    # Define a class named LifeSprites that inherits from the Spritesheet class
    def __init__(self, numlives):
        # Constructor method to initialize an instance of the LifeSprites class
        Spritesheet.__init__(self)
        # Call the constructor of the parent class (Spritesheet) to initialize inherited attributes
        self.resetLives(numlives)
        # Call the resetLives method with the numlives parameter to initialize the instance attribute 'images'

    def removeImage(self):
        # Define a method named removeImage
        if len(self.images) > 0:
            # Check if the length of the 'images' list is greater than 0
            self.images.pop(0)
            # Remove the first element from the 'images' list
```

```
    def resetLives(self, numlives):
        # Define a method named resetLives which takes numlives as input
        self.images = []
        # Initialize the instance attribute 'images' as an empty list
        for i in range(numlives):
            # Iterate over a range from 0 to numlives
            self.images.append(self.getImage(0,0))
            # Append an image obtained from calling the getImage method with coordinates (0, 0) to the 'images' list
    def getImage(self, x, y):
        return Spritesheet.getImage(self, x, y, 2*TILEWIDTH, 2*TILEHEIGHT)# Define a method named getImage which takes x and y coordinates as input
        # Call the getImage method of the parent class (Spritesheet) with adjusted coordinates and return the result
class MazeSprites(Spritesheet): # Define a class named MazeSprites that inherits from the Spritesheet class
    def __init__(self, mazefile, rotfile):
        # Constructor method to initialize an instance of the MazeSprites class
        Spritesheet.__init__(self)
        # Call the constructor of the parent class (Spritesheet) to initialize inherited attributes
        self.data = self.readMazeFile(mazefile)
        # Assign the result of calling the readMazeFile method with the mazefile parameter to the instance attribute 'data'
        self.rotdata = self.readMazeFile(rotfile)
        # Assign the result of calling the readMazeFile method with the rotfile parameter to the instance attribute 'rotdata'
    def getImage(self, x, y):# Define a method named getImage which takes x and y coordinates as input
        return Spritesheet.getImage(self, x, y, TILEWIDTH, TILEHEIGHT)
        # Call the getImage method of the parent class (Spritesheet) with provided coordinates and return the result
    def readMazeFile(self, mazefile):# Define a method named readMazeFile which takes mazefile as input
        return np.loadtxt(mazefile, dtype='<U1')
        # Load a text file using NumPy and return the result as a numpy array of Unicode strings
    def constructBackground(self, background, y):# Define a method named constructBackground which takes background and y as input
        for row in list(range(self.data.shape[0])):
            # Iterate over each row index in the 'data' attribute
            for col in list(range(self.data.shape[1])):
                # Iterate over each column index in the 'data' attribute
                if self.data[row][col].isdigit():# Check if the current element in 'data' is a digit
                    x = int(self.data[row][col]) + 12# Convert the digit to an integer and add 12 to get the sprite index
                    sprite = self.getImage(x, y)# Get the sprite image using the obtained index and provided y coordinate
                    rotval = int(self.rotdata[row][col])# Obtain rotation value from the corresponding element in 'rotdata'
                    sprite = self.rotate(sprite, rotval)# Rotate the sprite image based on the rotation value
                    background.blit(sprite, (col*TILEWIDTH, row*TILEHEIGHT)) # Draw the rotated sprite onto the background surface at the appropriate position
                elif self.data[row][col] == '=':
                    # Check if the current element in 'data' is '='
                    sprite = self.getImage(10, 8)
                    # Get a specific sprite image for '='
                    background.blit(sprite, (col*TILEWIDTH, row*TILEHEIGHT))
                    # Draw the sprite onto the background surface at the appropriate position
        return background
        # Return the modified background surface
    def rotate(self, sprite, value):
        # Define a method named rotate which takes a sprite and a rotation value as input
        return pygame.transform.rotate(sprite, value*90)
        # Rotate the sprite image by the specified angle (multiple of 90 degrees) using pygame's transformation function and return the result
```

TEXT.PY

```python
import pygame  # Importing the pygame library for game development.
from vector import Vector2  # Importing a Vector2 class from a custom module.
from constants import *  # Importing constants from another module.

class Text(object):
    def __init__(self, text, color, x, y, size, time=None, id=None, visible=True):
        self.id = id  # Assigning an id to the text object.
        self.text = text  # Storing the text content.
        self.color = color  # Storing the color of the text.
        self.size = size  # Storing the font size of the text.
        self.visible = visible  # Storing the visibility status of the text.
        self.position = Vector2(x, y)  # Creating a Vector2 object to store the position of the text.
        self.timer = 0  # Initializing a timer for the text.
        self.lifespan = time  # Storing the lifespan of the text.
        self.label = None  # Initializing the label attribute.
        self.destroy = False  # Flag to indicate if the text should be destroyed.
        self.setupFont("PressStart2P-Regular.ttf")  # Setting up the font for the text.
        self.createLabel()  # Creating the label for the text.

    def setupFont(self, fontpath):
        self.font = pygame.font.Font(fontpath, self.size)  # Loading the font for the text.

    def createLabel(self):
        self.label = self.font.render(self.text, 1, self.color)  # Rendering the text to create the label.

    def setText(self, newtext):
        self.text = str(newtext)  # Setting new text content.
        self.createLabel()  # Recreating the label with the new text.

    def update(self, dt):
        if self.lifespan is not None:  # Checking if the text has a lifespan.
            self.timer += dt  # Incrementing the timer.
            if self.timer >= self.lifespan:  # Checking if the text's lifespan has expired.
                self.timer = 0  # Resetting the timer.
                self.lifespan = None  # Disabling the lifespan.
                self.destroy = True  # Marking the text for destruction.

    def render(self, screen):
        if self.visible:  # Checking if the text is visible.
            x, y = self.position.asTuple()  # Extracting position coordinates.
            screen.blit(self.label, (x, y))  # Rendering the text label on the screen.


class TextGroup(object):
    def __init__(self):
        self.nextid = 10  # Initializing the next available id for text objects.
        self.alltext = {}  # Dictionary to store all text objects.
        self.setupText()  # Setting up text objects.
        self.showText(READYTXT)  # Showing the "READY" text at the beginning.
```

```python
    def addText(self, text, color, x, y, size, time=None, id=None):
        self.nextid += 1  # Incrementing the id for the next text object.
        self.alltext[self.nextid] = Text(text, color, x, y, size, time=time, id=id)  # Creating a new text object.
        return self.nextid  # Returning the id of the newly added text object.

    def removeText(self, id):
        self.alltext.pop(id)  # Removing a text object with a specific id from the collection.

    def setupText(self):
        size = TILEHEIGHT  # Setting the font size for text objects.
        # Creating various predefined text objects.
        self.alltext[SCORETXT] = Text("0".zfill(8), WHITE, 0, TILEHEIGHT, size)
        self.alltext[LEVELTXT] = Text(str(0).zfill(3), WHITE, 23*TILEWIDTH, TILEHEIGHT, size)
        self.alltext[READYTXT] = Text("READY!", YELLOW, 11.25*TILEWIDTH, 20*TILEHEIGHT, size, visible=False)
        self.alltext[PAUSETXT] = Text("PAUSED!", YELLOW, 10.625*TILEWIDTH, 20*TILEHEIGHT, size, visible=False)
        self.alltext[GAMEOVERTXT] = Text("GAMEOVER!", YELLOW, 10*TILEWIDTH, 20*TILEHEIGHT, size, visible=False)
        # Adding additional text objects for "SCORE" and "LEVEL".
        self.addText("SCORE", WHITE, 0, 0, size)
        self.addText("LEVEL", WHITE, 23*TILEWIDTH, 0, size)

    def update(self, dt):
        for tkey in list(self.alltext.keys()):  # Iterating through all text objects.
            self.alltext[tkey].update(dt)  # Updating each text object.
            if self.alltext[tkey].destroy:  # Checking if a text object needs to be destroyed.
                self.removeText(tkey)  # Removing the text object.

    def showText(self, id):
        self.hideText()  # Hiding all text objects.
        self.alltext[id].visible = True  # Showing a specific text object.

    def hideText(self):
        # Hiding specific text objects.
        self.alltext[READYTXT].visible = False
        self.alltext[PAUSETXT].visible = False
        self.alltext[GAMEOVERTXT].visible = False

    def updateScore(self, score):
        self.updateText(SCORETXT, str(score).zfill(8))  # Updating the score text object.

    def updateLevel(self, level):
        self.updateText(LEVELTXT, str(level + 1).zfill(3))  # Updating the level text object.

    def updateText(self, id, value):
        if id in self.alltext.keys():  # Checking if the specified text object exists.
            self.alltext[id].setText(value)  # Updating the text content.

    def render(self, screen):
        for tkey in list(self.alltext.keys()):  # Iterating through all text objects.
            self.alltext[tkey].render(screen)  # Rendering each text object.
```

PAUSER.PY

```python
# Class for managing pauses
class Pause(object):
     # Initialization
    def __init__(self, paused=False):
        self.paused = paused  # Sets the initial pause state
        self.timer = 0   # Timer starts at 0
        self.pauseTime = None  # No specific pause duration set by default
        self.func = None     # No function to execute at the end of a pause by default


    def update(self, dt):   #Updates the Pause object's state.
        if self.pauseTime is not None:   #If a timed pause is active
            self.timer += dt    #Increment the timer
            if self.timer >= self.pauseTime:    # Check if the pause duration has been reached
                self.timer = 0   # Reset timer
                self.paused = False   # Unpause the game
                self.pauseTime = None   # Clear pause duration
                return self.func   # Return the function to execute (if any)
        return None   # No function to execute if no timed pause is active

# Sets a new pause state.
    def setPause(self, playerPaused=False, pauseTime=None, func=None):
        self.timer = 0  # Reset timer
        self.func = func
        self.pauseTime = pauseTime   # Set pause duration (if any)
        self.flip()   # Toggle the paused state


    def flip(self):    # Inverts the paused state
        self.paused = not self.paused
```

VECTOR.PY

```python
import math

class Vector2(object):    #Args:x (float, optional): X-component of the vector.#y (float, optional): Y-component of the vector. Defaults to 0.


    def __init__(self, x=0, y=0):Initializes a Vector2 object.

        self.x = x  # Stores the x-component of the vector
        self.y = y   # Stores the y-component of the vector
        self.thresh = 0.000001

    def __add__(self, other):# Overloads the addition operator (+) to perform vector addition
        return Vector2(self.x + other.x, self.y + other.y)

    def __sub__(self, other): #Overloads the subtraction operator (-) to perform vector subtraction.
        return Vector2(self.x - other.x, self.y - other.y)

    def __neg__(self):  #Overloads the negation operator (-) to return the negative of the vector
        return Vector2(-self.x, -self.y)

    def __mul__(self, scalar):  #Overloads the multiplication operator (*) to perform scalar multiplication.
        return Vector2(self.x * scalar, self.y * scalar)

    def __div__(self, scalar):  #Overloads the true division operator (/) to perform vector division by a scalar.
        if scalar != 0:
            return Vector2(self.x / float(scalar), self.y / float(scalar))
        return None

    def __truediv__(self, scalar):#Overloads the true division operator (/) to perform vector division by a scalar.
        return self.__div__(scalar)

    def __eq__(self, other):   #Overloads the equality operator (==) to perform approximate vector equality check.
        if abs(self.x - other.x) < self.thresh:
            if abs(self.y - other.y) < self.thresh:
                return True
        return False

    def magnitudeSquared(self):
        return self.x**2 + self.y**2

    def magnitude(self):
        return math.sqrt(self.magnitudeSquared())

        #Calculates the magnitude (length) of the vector.

        #Returns:
            #float: The magnitude of the vector.

    def copy(self):
        return Vector2(self.x, self.y)

    def asTuple(self):  #Returns the vector components as a tuple (x, y).


    def asInt(self):
        return int(self.x), int(self.y)

    def __str__(self):
        return "<"+str(self.x)+", "+str(self.y)+">"
```

FRUIT.PY

```python
# Import necessary libraries
import pygame
from entity import Entity   # Import the base Entity class
from constants import *    # Import all constants from the constants module
from sprites import FruitSprites   # Import the FruitSprites class

class Fruit(Entity):  # Define the Fruit class, inheriting from Entity
    def __init__(self, node, level=0): # Constructor to initialize the Fruit object
        Entity.__init__(self, node)
        # Set the fruit's attributes
        self.name = FRUIT # Name of the fruit
        self.color = GREEN # Default color of the fruit
        self.lifespan = 5 # Lifespan of the fruit in seconds
        self.timer = 0  # Internal timer to track lifespan
        self.destroy = False  #indicating if the fruit should be destroyed
        self.points = 100 + level*20  # Points awarded for eating the fruit (increases with level)
        self.setBetweenNodes(RIGHT)  # Set initial movement direction
        self.sprites = FruitSprites(self, level) # Create fruit sprites based on level

    def update(self, dt):   # Update method to handle fruit behavior each frame
        self.timer += dt #Update the internal timer by elapsed time (dt)
        if self.timer >= self.lifespan:    # Check if the timer has reached the lifespan
            self.destroy = True   # Set the destroy flag to True for removal
```

ANIMATION.PY

```python
from constants import *  # Import all constants from the constants module

class Animator(object): # This class manages the animation of a sprite sheet by keeping track of frames, speed, and looping behavior.
    def __init__(self, frames=[], speed=20, loop=True):
        self.frames = frames  # List of animation frames (images)
        self.current_frame = 0  # Index of the current frame
        self.speed = speed    # Animation speed (frames per second)
        self.loop = loop    # Whether to loop the animation
        self.dt = 0  # Time elapsed since the last frame change (internal timer)
        self.finished = False # indicating if the animation has finished playing

    def reset(self): # Resets the animation to the first frame and sets the finished flag to False.
        self.current_frame = 0
        self.finished = False

    def update(self, dt):   #Updates the animation based on the elapsed time (dt).
        if not self.finished:
            self.nextFrame(dt)  # Update the current frame if not finished
        if self.current_frame == len(self.frames):  # Check for animation completion and handle looping or setting finished flag
            if self.loop:
                self.current_frame = 0
            else:
                self.finished = True
                self.current_frame -= 1

        return self.frames[self.current_frame]# Return the current frame image

    def nextFrame(self, dt):   #Updates the internal timer (dt) and advances to the next frame if enough time has passed.
        self.dt += dt
        if self.dt >= (1.0 / self.speed):
            self.current_frame += 1
            self.dt = 0 # Reset timer for the next frame
```

MAZEDATA.PY

```
from constants import *  # Import all constants from the constants module

class MazeBase(object):  #This class defines the base structure for representing mazes in the game.
    def __init__(self):  # Initializes the MazeBase object.
        self.portalPairs = {}  # Dictionary to store portal connections
        self.homeoffset = (0, 0)  # Offset for the "home" area of the maze (where Pacman starts)
        self.ghostNodeDeny = {UP:(), DOWN:(), LEFT:(), RIGHT:()}   # Dictionary specifying directions (UP, DOWN, LEFT, RIGHT) and node coordinates where ghosts are denied access

    def setPortalPairs(self, nodes):  #Sets portal connections in the maze data structure (nodes object).
        for pair in list(self.portalPairs.values()):  # Iterate through portal pairs and call the nodes object's method to set them in the maze
            nodes.setPortalPair(*pair)  # Unpack the pair for the nodes object's method

    def connectHomeNodes(self, nodes):  #Connects the "home" nodes of the maze to the rest of the maze structure.

        key = nodes.createHomeNodes(*self.homeoffset) # Create the home nodes with the specified offset
        nodes.connectHomeNodes(key, self.homenodeconnectLeft, LEFT)  # Connect the home nodes to the left and right sides using provided connection functions
        nodes.connectHomeNodes(key, self.homenodeconnectRight, RIGHT)

    def addOffset(self, x, y): # Calculates a new coordinate by adding the home area offset to the given x and y values.
        return x+self.homeoffset[0], y+self.homeoffset[1]   # Add the home area offset to the given coordinates

    def denyGhostsAccess(self, ghosts, nodes):  #Restricts access for ghosts in specific areas of the maze.
        # Deny access for ghosts in LEFT and RIGHT directions at offset positions relative to home area
        nodes.denyAccessList(*(self.addOffset(2, 3) + (LEFT, ghosts)))
        nodes.denyAccessList(*(self.addOffset(2, 3) + (RIGHT, ghosts)))

        for direction in list(self.ghostNodeDeny.keys()):   # Iterate through directions and their denied node coordinates in ghostNodeDeny
            for values in self.ghostNodeDeny[direction]:
                # Deny access for ghosts in the specified direction at the listed node coordinates

                nodes.denyAccessList(*(values + (direction, ghosts)))


class Maze1(MazeBase):  #This class defines a specific maze layout named "maze1" that inherits from MazeBase.
    def __init__(self):
        MazeBase.__init__(self)  # Call the base class constructor to inherit common properties
        self.name = "maze1"   # Name of the maze
        self.portalPairs = {0:((0, 17), (27, 17))}  # Define portal connections for this maze
        self.homeoffset = (11.5, 14)   # Set the home area offset for this maze (where Pacman starts)
        self.homenodeconnectLeft = (12, 14)   # Node to connect home area to the left side
        self.homenodeconnectRight = (15, 14)   # Node to connect home area to the right side
        self.pacmanStart = (15, 26)  # Starting position for Pacman in this maze
        self.fruitStart = (9, 20)  # Starting position for fruits in this maze
        # Define areas where ghosts are denied access in this maze:
        self.ghostNodeDeny = {UP:((12, 14), (15, 14), (12, 26), (15, 26)), LEFT:(self.addOffset(2, 3),),
                              RIGHT:(self.addOffset(2, 3),)}

class Maze2(MazeBase):  # This class defines a specific maze layout named "maze2" that inherits from MazeBase.
    def __init__(self):  #Initializes the Maze2 object with specific properties for the "maze2" layout.
        MazeBase.__init__(self)   # Call the base class constructor to inherit common properties
        self.name = "maze2"
        # Define properties specific to the maze2 layout (similar structure to Maze1):
        self.portalPairs = {0:((0, 4), (27, 4)), 1:((0, 26), (27, 26))}
        self.homeoffset = (11.5, 14)
        self.homenodeconnectLeft = (9, 14)
        self.homenodeconnectRight = (18, 14)
        self.pacmanStart = (16, 26)
        self.fruitStart = (11, 20)
        self.ghostNodeDeny = {UP:((9, 14), (18, 14), (11, 23), (16, 23)), LEFT:(self.addOffset(2, 3),),
                              RIGHT:(self.addOffset(2, 3),)}


class MazeData(object):
    def __init__(self):
        self.obj = None
        self.mazedict = {0:Maze1, 1:Maze2}

    def loadMaze(self, level):
        self.obj = self.mazedict[level%len(self.mazedict)]()
```

PELLETS.PY

```python
import pygame
from vector import Vector2
from constants import *
import numpy as np

class Pellet(object):  # Class representing a single pellet on the map
    def __init__(self, row, column):
        self.name = PELLET
        #Calculate the position of the pellet based on row, column, and tile dimensions
        self.position = Vector2(column*TILEWIDTH, row*TILEHEIGHT)
        self.color = WHITE   # Set the color of the pellet (white based on the code)
        self.radius = int(2 * TILEWIDTH / 16) # Define the radius of the pellet
        self.collideRadius = 2 * TILEWIDTH / 16 # Define the collision radius
        self.points = 10 # Set the number of points awarded for eating the pellet
        self.visible = True  #indicating whether the pellet is currently visible

    def render(self, screen):  # Function to draw the pellet on the Pygame screen
        if self.visible:  #Check if the pellet is visible before drawing
            adjust = Vector2(TILEWIDTH, TILEHEIGHT) / 2
            p = self.position + adjust  #Calculate the adjusted center position for drawing the circle
            pygame.draw.circle(screen, self.color, p.asInt(), self.radius)  #Use Pygame functions to draw a circle representing the pellet

#Class representing a power pellet, inheriting from the base Pellet class
class PowerPellet(Pellet):
    def __init__(self, row, column):
        Pellet.__init__(self, row, column)
        self.name = POWERPELLET #Set the name of the power pellet
        self.radius = int(8 * TILEWIDTH / 16)  #define a larger radius for the power pellet compared to a regular pellet
        self.points = 50  #Set the number of points awarded for eating a power pellet
        self.flashTime = 0.2 # Define the flash time interval for the power pellet
        self.timer= 0  Internal timer used to track the flashing behavior

    def update(self, dt):  #Function to update the power pellet's visibility
        self.timer += dt  # Update the internal timer by adding the elapsed time (dt)
        if self.timer >= self.flashTime:   # Check if the timer has reached the flash time
            self.visible = not self.visible  # Flip the visibility flag (on/off) to create a flashing effect
            self.timer = 0   # Reset the timer for the next flash cycle


class PelletGroup(object):  # Class to manage a collection of pellet and power pellet objects
    def __init__(self, pelletfile):
        self.pelletList = []   # Create empty lists to store pellets and power pellets
        self.powerpellets = []
        self.createPelletList(pelletfile)# Call a function to create the pellet list from a data file
        self.numEaten = 0  # Initialize the number of eaten pellets to 0


    def update(self, dt):  # Function to update the power pellets in the group
        for powerpellet in self.powerpellets: # Loop through all power pellets in the list
            powerpellet.update(dt)

    def createPelletList(self, pelletfile):  #Function to create the pellet list by reading a data file
        data = self.readPelletfile(pelletfile)
         #Loop through each row and column of the data
        for row in range(data.shape[0]):
            for col in range(data.shape[1]):
                if data[row][col] in ['.', '+']:
                    self.pelletList.append(Pellet(row, col))
                elif data[row][col] in ['P', 'p']:
                    # Create a PowerPellet object and add it to both pelletList and powerpellets list
                    pp = PowerPellet(row, col)
                    self.pelletList.append(pp)
                    self.powerpellets.append(pp)

    def readPelletfile(self, textfile):
        return np.loadtxt(textfile, dtype='<U1')

    def isEmpty(self):
        if len(self.pelletList) == 0:
            return True
        return False

    def render(self, screen):
        for pellet in self.pelletList:
            pellet.render(screen)
```

CONSTANTS.PY

```
# Define tile dimensions in pixels
TILEWIDTH = 16
TILEHEIGHT = 16
# Define the number of rows and columns in the maze layout
NROWS = 36
NCOLS = 28
# Calculate the screen width and height based on tile dimensions and number of tiles
SCREENWIDTH = NCOLS*TILEWIDTH
SCREENHEIGHT = NROWS*TILEHEIGHT
SCREENSIZE = (SCREENWIDTH, SCREENHEIGHT)
# Define colors used in the game (RGB values)
BLACK = (0, 0, 0)
YELLOW = (255, 255, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
PINK = (255,100,150)
TEAL = (100,255,255)
ORANGE = (230,190,40)
GREEN = (0, 255, 0)
# Define colors used in the game (RGB values)
STOP = 0
UP = 1
DOWN = -1
LEFT = 2
RIGHT = -2
PORTAL = 3
# Define constants for game entities
PACMAN = 0
PELLET = 1
POWERPELLET = 2
GHOST = 3
BLINKY = 4
PINKY = 5
INKY = 6
CLYDE = 7
FRUIT = 8
# Define constants for ghost behavior states
SCATTER = 0
CHASE = 1
FREIGHT = 2
SPAWN = 3
#Define tile dimensions in pixels for consistency
SCORETXT = 0TILEWIDTH = 16
TILEHEIGHT = 16
#Define the number of rows and columns in the maze layout
NROWS = 36
NCOLS = 28
# Calculate the screen width and height based on tile dimensions and number of tiles
SCREENWIDTH = NCOLS*TILEWIDTH
SCREENHEIGHT = NROWS*TILEHEIGHT
SCREENSIZE = (SCREENWIDTH, SCREENHEIGHT)
# Define colors used in the game (RGB values)
BLACK = (0, 0, 0)
YELLOW = (255, 255, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
PINK = (255,100,150)
TEAL = (100,255,255)
ORANGE = (230,190,40)
GREEN = (0, 255, 0)
# Define colors used in the game (RGB values)
STOP = 0
UP = 1
DOWN = -1
LEFT = 2
RIGHT = -2
PORTAL = 3
# Define constants for game entities
PACMAN = 0
PELLET = 1
POWERPELLET = 2
GHOST = 3
BLINKY = 4
PINKY = 5
INKY = 6
CLYDE = 7
FRUIT = 8
# Define constants for ghost behavior states
SCATTER = 0
CHASE = 1
FREIGHT = 2
SPAWN = 3

SCORETXT = 0  # Identifier for the score text element
LEVELTXT = 1  # Identifier for the level text element
READYTXT = 2  # Identifier for the "Ready" text element (displayed before game start)
PAUSETXT = 3  # Identifier for the "Pause" text element (shown when the game is paused)
GAMEOVERTXT = 4  # Identifier for the "Game Over" text element
```

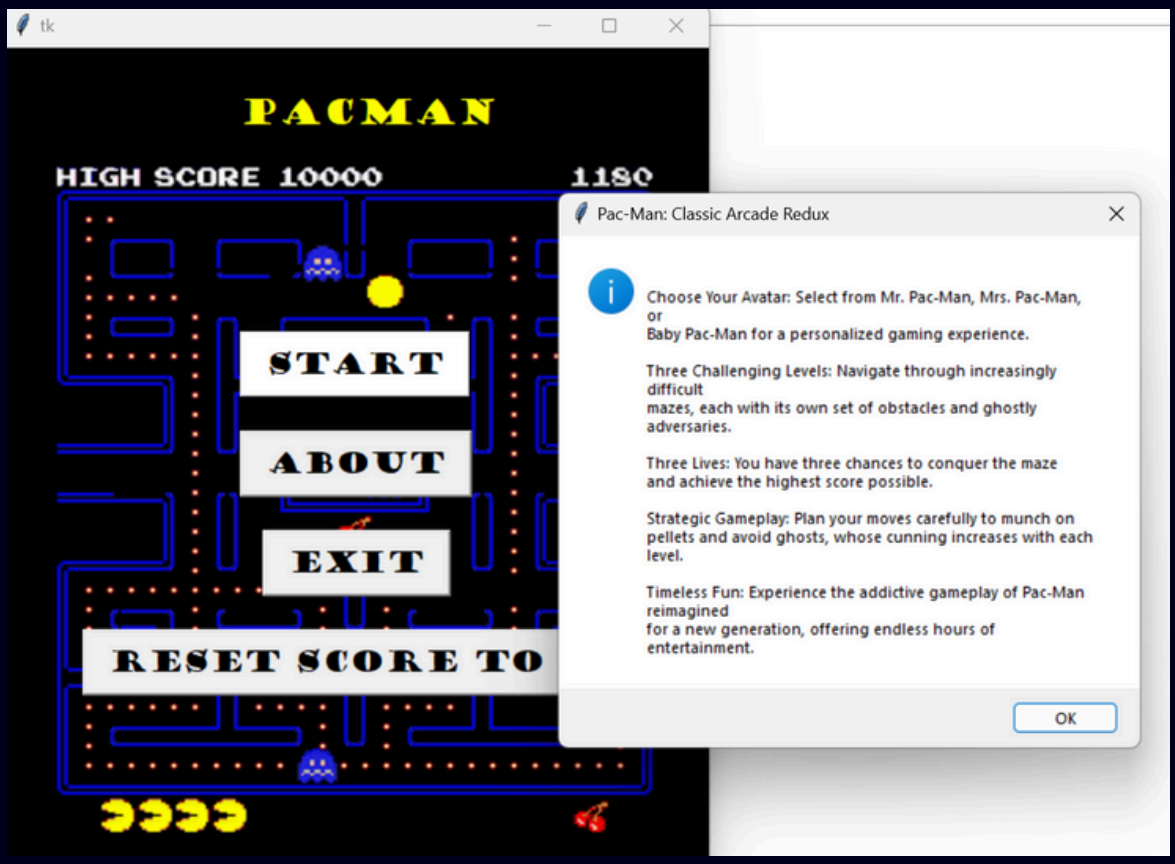
NODES.PY

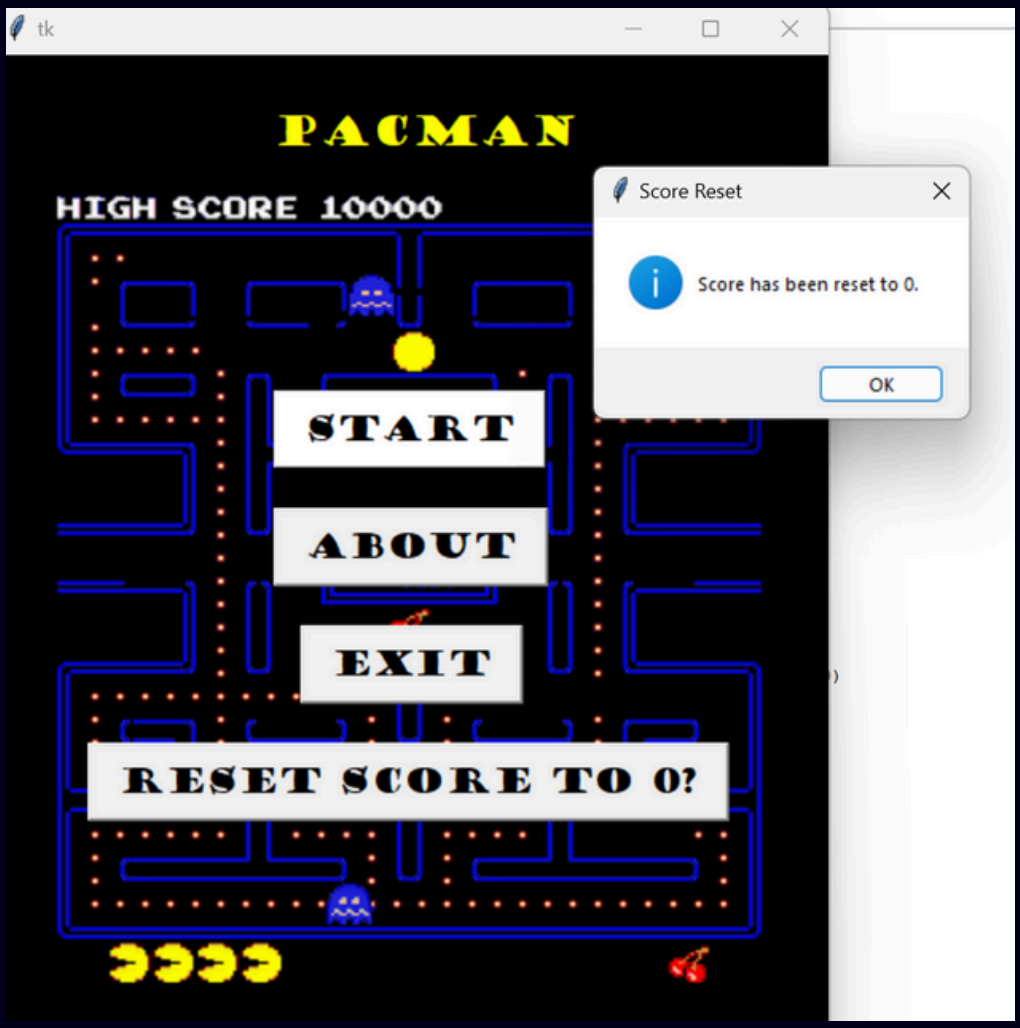
```python
import pygame  # Import necessary libraries
from vector import Vector2  # a custom class for representing 2D vector
from constants import *  #contains constants used throughout the game
import numpy as np  # For numerical operations

class Node(object): # Define the Node class representing a single node in the maze
    def __init__(self, x, y):  # Initializes a Node object with its position and empty neighbor/access information.

        self.position = Vector2(x, y)  # Position of the node
        # Dictionary to store neighboring nodes in different directions
        self.neighbors = {UP:None, DOWN:None, LEFT:None, RIGHT:None, PORTAL:None}
         # Dictionary to store access permissions for different entities in each direction
        self.access = {UP:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT],
                       DOWN:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT],
                       LEFT:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT],
                       RIGHT:[PACMAN, BLINKY, PINKY, INKY, CLYDE, FRUIT]}

    def denyAccess(self, direction, entity):  #Denies access for a specific entity
        if entity.name in self.access[direction]:
            self.access[direction].remove(entity.name)    # Remove entity from allowed list

    def allowAccess(self, direction, entity):  # Grants access for a specific entity in a particular direction from this node.
        if entity.name not in self.access[direction]:
            self.access[direction].append(entity.name)    # Add entity to allowed list

    def render(self, screen):  # Renders the node visually on the screen (likely for debugging or visualization purposes).
        for n in self.neighbors.keys():
            if self.neighbors[n] is not None:
                # Draw lines connecting this node to its neighbors (if they exist)
                line_start = self.position.asTuple()
                line_end = self.neighbors[n].position.asTuple()
                pygame.draw.line(screen, WHITE, line_start, line_end, 4)
                # Draw a red circle at the node's position
                pygame.draw.circle(screen, RED, self.position.asInt(), 12)

# Define the NodeGroup class representing the entire maze structure
class NodeGroup(object):
    #This class manages a group of nodes representing the entire maze layout. It reads the maze data from a text file, creates the node connections, and provides functionalities for access

    def __init__(self, level):
        self.level = level
        self.nodesLUT = {}   # Dictionary to store nodes with their coordinates as keys (likely for efficient lookup)
        self.nodeSymbols = ['+', 'P', 'n']  # Symbols used to represent different elements in the maze text file
        self.pathSymbols = ['.', '-', '|', 'p']# Symbols used to represent
        data = self.readMazeFile(level)
        self.createNodeTable(data)
        self.connectHorizontally(data)
        self.connectVertically(data)
        self.homekey = None

    def readMazeFile(self, textfile):
        return np.loadtxt(textfile, dtype='<U1')

    def createNodeTable(self, data, xoffset=0, yoffset=0):
        for row in list(range(data.shape[0])):
            for col in list(range(data.shape[1])):
                if data[row][col] in self.nodeSymbols:
                    x, y = self.constructKey(col+xoffset, row+yoffset)
                    self.nodesLUT[(x, y)] = Node(x, y)

    def constructKey(self, x, y):  #Creates a unique key for a node based on its coordinates.
        # Use constants.TILEWIDTH to convert x, y to pixel coordinates
        return x * TILEWIDTH, y * TILEHEIGHT


    def connectHorizontally(self, data, xoffset=0, yoffset=0):
        #Establishes horizontal connections between nodes based on maze data.
        for row in list(range(data.shape[0])): # Iterate through rows of maze data
            key = None
            for col in list(range(data.shape[1])): # Iterate through rows of maze data
                if data[row][col] in self.nodeSymbols:  # Check if current cell represents a node
                    if key is None:
                        # If it's the first node in the row, create a key for it
                        key = self.constructKey(col+xoffset, row+yoffset)
                    else:
                        # If there's a previous node, connect them horizontally
                        otherkey = self.constructKey(col+xoffset, row+yoffset)
                        self.nodesLUT[key].neighbors[RIGHT] = self.nodesLUT[otherkey]
                        self.nodesLUT[otherkey].neighbors[LEFT] = self.nodesLUT[key]
                        key = otherkey  # Update key for the next node in the row
                elif data[row][col] not in self.pathSymbols:
                    key = None


    def connectVertically(self, data, xoffset=0, yoffset=0): # Establishes vertical connections between nodes based on maze data.

        dataT = data.transpose()  # Transpose the data for processing columns as rows
        for col in list(range(dataT.shape[0])): #  Iterate through transposed data (columns)
            key = None
            # Iterate through rows of transposed data
            for row in list(range(dataT.shape[1])):
                if dataT[col][row] in self.nodeSymbols: # Check if current cell represents a node
                    if key is None:
                        # If it's the first node in the column, create a key for it
                        key = self.constructKey(col+xoffset, row+yoffset)
                    else:
                        # If there's a previous node, connect them vertically
                        otherkey = self.constructKey(col+xoffset, row+yoffset)
                        self.nodesLUT[key].neighbors[DOWN] = self.nodesLUT[otherkey]
                        self.nodesLUT[otherkey].neighbors[UP] = self.nodesLUT[key]
                        key = otherkey
                elif dataT[col][row] not in self.pathSymbols:
                    # If it's not a path or node symbol, reset key
                    key = None


    def getStartTempNode(self):
        #Returns a temporary starting node from the maze (likely for initial setup or testing).
        nodes = list(self.nodesLUT.values())
        return nodes[0] # Return the first node in the maze for initial purposes

    def setPortalPair(self, pair1, pair2):  #Creates a portal connection between two nodes.
        key1 = self.constructKey(*pair1)# Unpack tuple to get individual coordinates for key1
        key2 = self.constructKey(*pair2)# Unpack tuple to get individual coordinates for key2
        if key1 in self.nodesLUT.keys() and key2 in self.nodesLUT.keys():
            self.nodesLUT[key1].neighbors[PORTAL] = self.nodesLUT[key2]
            self.nodesLUT[key2].neighbors[PORTAL] = self.nodesLUT[key1]
            # Set both nodes' PORTAL neighbor to the other node


    def createHomeNodes(self, xoffset, yoffset):
        #Creates a separate home area for entities (like Pacman's base) using a predefined layout.
        # Predefined home area layout using symbols
        homedata = np.array([['X','X','+','X','X'],
                             ['X','X','.','X','X'],
                             ['+','X','.','X','+'],
                             ['+','.','+','.','+'],
                             ['+','X','X','X','+']])

        self.createNodeTable(homedata, xoffset, yoffset)  # Create nodes from home area data
        self.connectHorizontally(homedata, xoffset, yoffset)   # Connect nodes horizontally
        self.connectVertically(homedata, xoffset, yoffset)   # Connect nodes vertically
        self.homekey = self.constructKey(xoffset+2, yoffset)   # Get key for the central node
        return self.homekey  # Return the key of the central home area node

    def connectHomeNodes(self, homekey, otherkey, direction):
        # Connects the home area to another part of the maze at a specific direction.

        key = self.constructKey(*otherkey)  # Unpack tuple to get individual coordinates for key
        self.nodesLUT[homekey].neighbors[direction] = self.nodesLUT[key]
```

```
        self.nodesLUT[key].neighbors[direction*-1] = self.nodesLUT[homekey]
        # Set the neighbors for both nodes based on the direction

    def getNodeFromPixels(self, xpixel, ypixel):
        #Retrieves a node object based on its pixel coordinates.
        if (xpixel, ypixel) in self.nodesLUT.keys():
            return self.nodesLUT[(xpixel, ypixel)]
        return None   # Return None if no node found at the coordinates

    def getNodeFromTiles(self, col, row):
        x, y = self.constructKey(col, row)
        if (x, y) in self.nodesLUT.keys():
            return self.nodesLUT[(x, y)]
        return None

    def denyAccess(self, col, row, direction, entity):
        node = self.getNodeFromTiles(col, row)
        if node is not None:
            node.denyAccess(direction, entity)

    def allowAccess(self, col, row, direction, entity):
        node = self.getNodeFromTiles(col, row)
        if node is not None:
            node.allowAccess(direction, entity)

    def denyAccessList(self, col, row, direction, entities):
        for entity in entities:
            self.denyAccess(col, row, direction, entity)

    def allowAccessList(self, col, row, direction, entities):
        for entity in entities:
            self.allowAccess(col, row, direction, entity)

    def denyHomeAccess(self, entity):
        self.nodesLUT[self.homekey].denyAccess(DOWN, entity)

    def allowHomeAccess(self, entity):
        self.nodesLUT[self.homekey].allowAccess(DOWN, entity)

    def denyHomeAccessList(self, entities):
        for entity in entities:
            self.denyHomeAccess(entity)

    def allowHomeAccessList(self, entities):
        for entity in entities:
            self.allowHomeAccess(entity)
            ----------------------------

    def render(self, screen):
        for node in self.nodesLUT.values():
            node.render(screen)
```

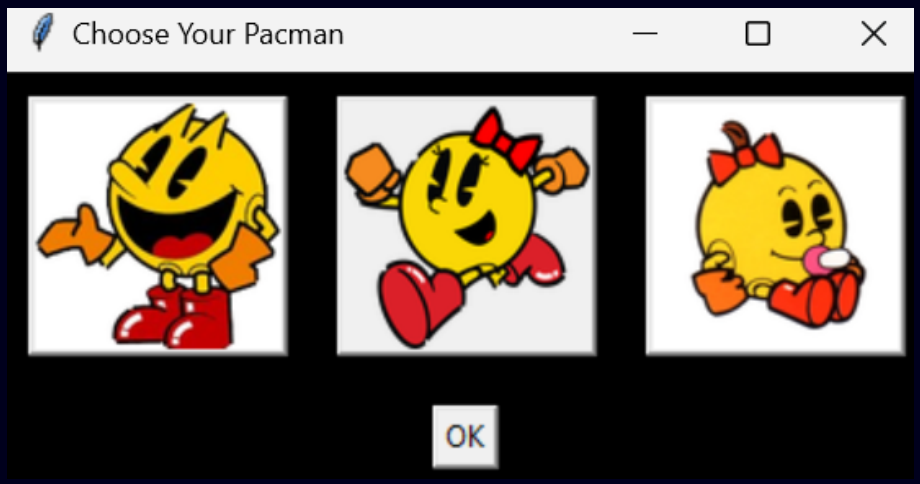
OUTPUT

START MENU


tk
PACMAN
HIGH SCORE 10000
1180
START
ABOUT
EXIT
RESET SCORE TO
Pac-Man: Classic Arcade Redux
Choose Your Avatar: Select from Mr. Pac-Man, Mrs. Pac-Man, or
Baby Pac-Man for a personalized gaming experience.
Three Challenging Levels: Navigate through increasingly difficult
mazes, each with its own set of obstacles and ghostly adversaries.
Three Lives: You have three chances to conquer the maze and achieve the highest score possible.
Strategic Gameplay: Plan your moves carefully to munch on pellets and avoid ghosts, whose cunning increases with each level.
Timeless Fun: Experience the addictive gameplay of Pac-Man reimagined
for a new generation, offering endless hours of entertainment.
OK


tk
PACMAN
HIGH SCORE 10000
START
ABOUT
EXIT
RESET SCORE TO 0?
Score Reset
Score has been reset to 0.
OK

CHOOSE YOUR PACMAN



MAIN GAME LEVEL 1

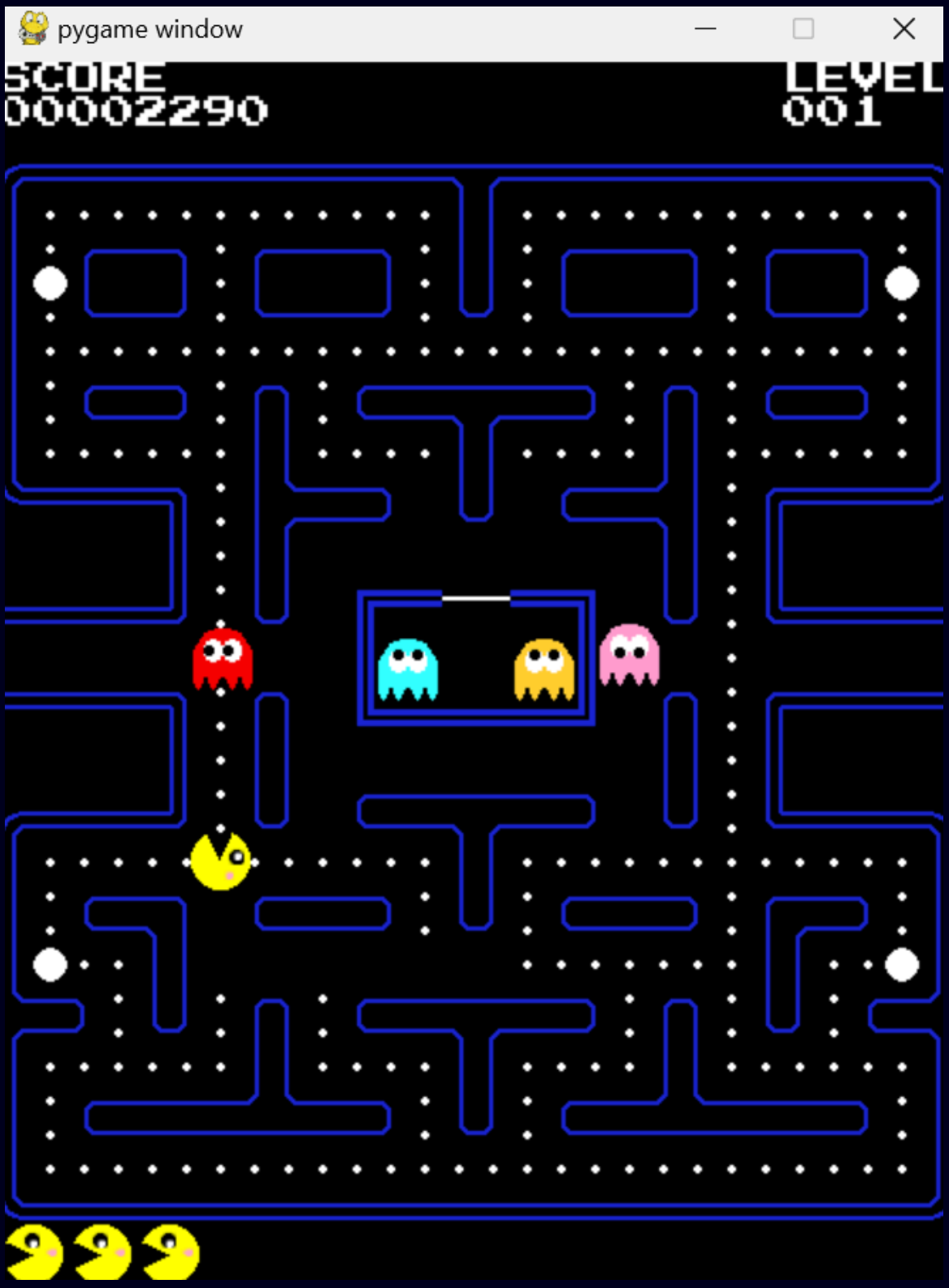
-MRS PACMAN

MAIN GAME LEVEL 1

-MR PACMAN

MAIN GAME LEVEL 1

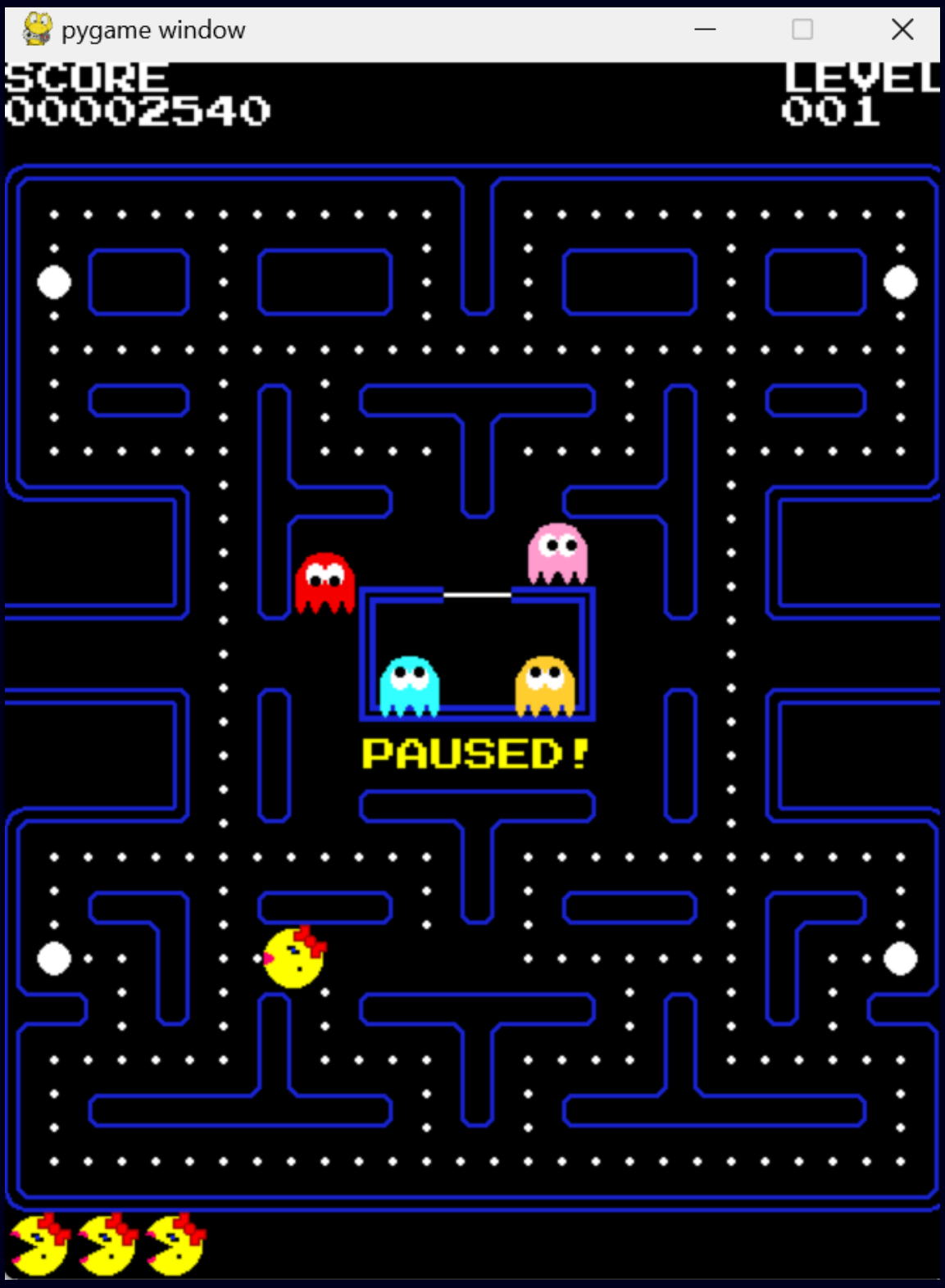
-BABY PACMAN



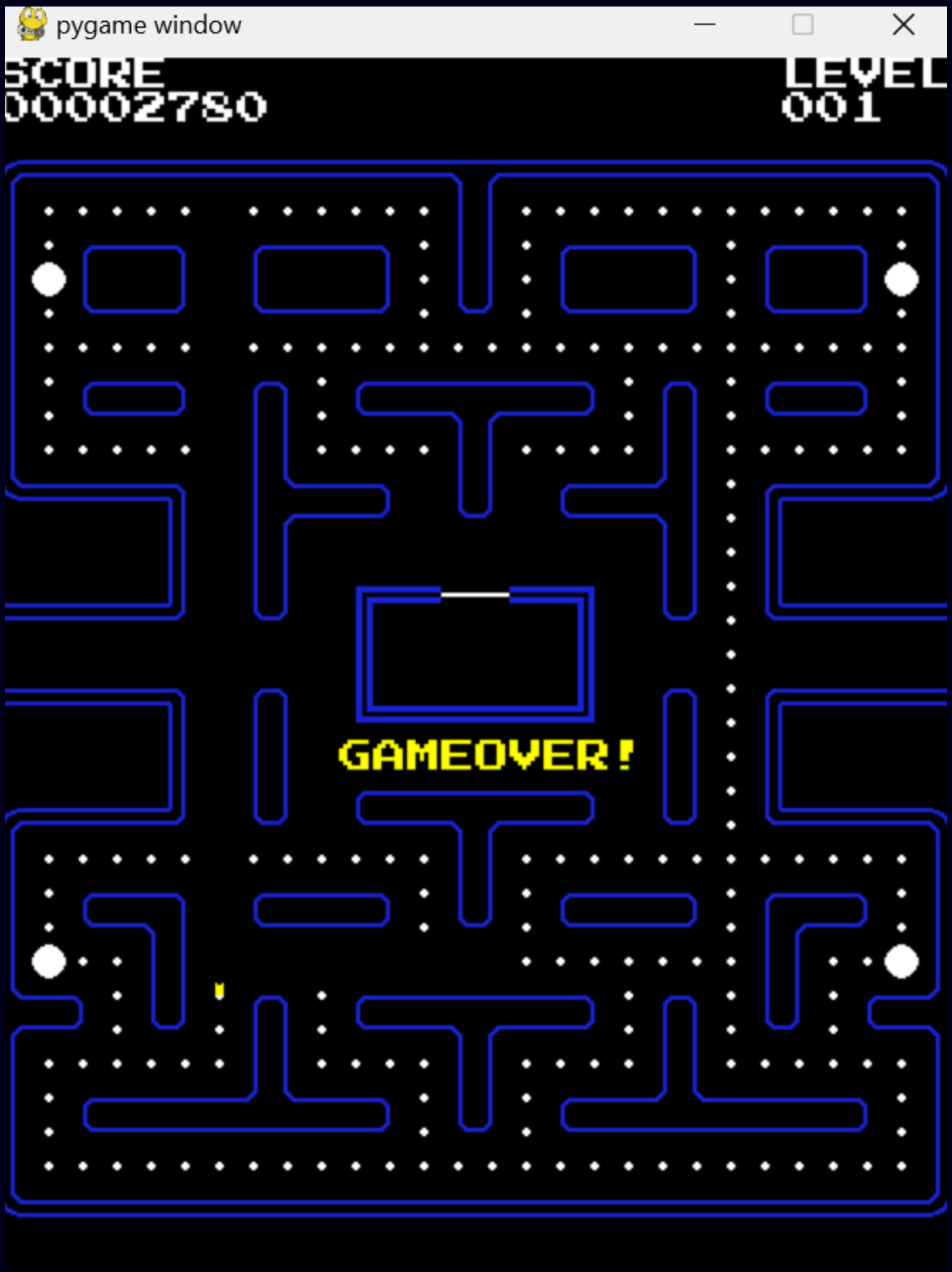
LEVEL 2

LEVEL 3

PAUSED SCREEN



GAME OVER SCREEN



BIBLIOGRAPHY

1. https://stackoverflow.com/
2. https://www.geeksforgeeks.org/pygame-tutorial/
3. https://www.pygame.org/docs/tut/MakeGames.html
4. https://www.pygame.org/tags/all
5. https://pythonprogramming.net/creating-main-menu-tkinter/
6. https://www.youtube.com/channel/UCV5Ab39YnXvTZ6Grar9URxQ
7. https://www.youtube.com/watch?v=9H27CimgPsQ&t=301s

REFLECTION

Developing the Pacman project using threading, Pygame, Tkinter, and object-oriented programming (OOP) has been an enlightening journey. This comprehensive approach allowed for a structured development process, dividing functionalities into separate modules and classes. Utilizing threading enhanced the game's responsiveness, ensuring smooth gameplay while maintaining concurrent tasks. Pygame's robust library facilitated efficient game development, providing essential functionalities for rendering graphics, handling events, and managing sound effects seamlessly. Integrating Tkinter for score display and OS for file operations added versatility to the project, enriching the user experience. Embracing OOP principles empowered modular design, enhancing code readability, scalability, and maintainability. Overall, this project provided valuable insights into leveraging diverse tools and methodologies to create engaging and functional game applications.