**Prompt**

*This latest code does not die but it also does not prioritize eating the apple and just makes the snake move in an area where it does not die or collide with itself or the blocks. This defeats the purpose of the game. Please fix it.*

**Reasoned about Snake AI and game behavior for 47 seconds**

**Analyzing the pathfinding code**

The code leverages a BFS algorithm for snake movement, ensuring it avoids obstacles and self-collision, which might hinder progress toward eating the apple.

**Shifting priorities**

OK, let me see. The code's AI chooses safe paths, skipping apples. To fix this, I'll tweak it to target apples first, avoiding death or collision risks.

**Tweaking BFS pathfinding**

Now working through BFS, I notice the tail's freedom isn't accounted for. This might mess with safety checks, so I'll refine the algorithm for precise results.

**Tweaking the strategy**

I found that the snake's head avoids the tail, leading it to safer paths and ignoring the apple. It's key to adjust the algorithm to help the snake prioritize the apple.

**Ensuring safety**

I’m thinking through a safety check where the snake prioritizes an apple if it’s safe to eat. This keeps the game engaging and prevents premature errors.

**Revising growth tactics**

I’m adjusting the algorithm to let the snake prioritize the apple when safe. This change helps avoid getting stuck by focusing on growth and direction instead of risk factors.

**Balancing snake growth**

OK, let's think it through: the snake's growth needs careful handling to keep safety checks accurate and to avoid ignoring the tail during BFS simulations.

**Balancing tail rules**

In simulation, treating the tail as free conflicts with the game rule where tail collision ends the game. Necessity might lead to a unique solution.

**Crafting the strategy**

I’m working through the "follow the tail" heuristic, carefully addressing the snake's growth and its tail, which remains in the same spot post-fruit consumption, ensuring a safe path to the tail.

**Key safety checks**

Ensuring the snake prioritizes apple eating while maintaining a free tail for risk management. This approach balances the game's rules with our safety simulation, focusing on growth and hazard avoidance.

**Ensuring safety**

I’m mapping out the simulation to prioritize the snake seeking fruit, but also need to make sure any path is safe, even without the tail moving, to avoid collision risks.

**Balancing safety**

I’m pondering whether the carrot and the snake's head should align for safety. Even if the tail stays static, I’m ensuring secure paths to the carrot.

**Balancing risk**

I’m adjusting the AI to aggressively pursue apples, even if it means encountering dangers. The goal is to find a good balance that keeps the snake alive while tracking the apple.

**Crafting the strategy**

I’m honing in on a BFS approach to prioritize apple eating, treating the tail as free if not consuming fruit. Safety moves are a fallback if a direct path is unavailable.

**Simulating with progress**

I’m tweaking the simulation to map out paths to fruit, toggling tail freedom for the final step if the fruit is still on the list. This keeps the options open.

**Weighing dangers**

Imagine letting the snake chase the fruit without safety checks, using collision detection to manage potential risks. This method tackles the problem head-on.

**Prioritizing fruit consumption**

The user's request hints at streamlining the simulation, especially if the next move is fruit. This adjustment enhances performance, aligning with the game's apple-eating goals.

**Reevaluating strategy**

I'm thinking through a simulation tweak that lets the snake prioritize apple consumption. This means assessing safety without considering the tail as an obstacle if the next move involves fruit.

**Built-in priorities**

I'm charting a course to let the snake always head straight for the fruit, balancing risk with efficiency by using BFS to find the best path.

**Crafting adaptive strategies**

I'm thinking through a plan for the snake to target the fruit first, using BFS to chart the safest route. If the fruit isn't reachable, it will follow safer moves back.

**Adjusting the strategy**

I'm considering the snake's tail as an obstacle in the BFS path to the fruit. This avoids collisions, ensuring the snake can safely pursue the apple.

**Pondering path planning**

I'm weighing whether to treat the snake's tail as an obstacle in BFS planning. This approach avoids collisions but complicates pathfinding, given the tail's dynamic movement.

**Crafting BFS paths**

I'm developing a BFS variant to pursue fruit, avoiding the tail cell. If the fruit isn't reachable, the snake will follow a safer path, avoiding collisions and minimizing tail interference.

**Adapting BFS strategy**

I'm adapting the BFS approach to treat the snake's tail as an obstacle, even though it might block the snake's path to the fruit, to strictly adhere to game rules.

**Prioritizing apple intake**

I’m working on adjusting the BFS to prioritize snake movements towards the apple, treating all snake segments, including the tail, as obstacles to ensure food safety.

**Assessing safety**

OK, let me see. I’m working through simulated safety by copying the snake and checking each cell in the path to the fruit, considering obstacles like the snake's tail and traps.

**Evaluating fruit consumption**

I'm weighing whether to allow the snake to eat the fruit and possibly risk retraction, factoring in game dynamics and the possibility of heading towards a dead-end path.

**Evaluating game strategy**

I'm thinking about removing the safety check to let the snake riskily go for the fruit. If a path is found, take it; otherwise, revert to following the tail.

**Evaluating strategies**

I'm considering whether to remove the safety simulation check to let the snake prioritize the apple over safety by default, but now I'm considering a twist if the snake isn't eating the fruit.

**Crafting the strategy**

I’m outlining a snake AI strategy: prioritize the apple if possible, and revert to tail-safe moves if needed. Removing the safety simulation check seems like the best approach for now.

**Tailoring the heuristic**

OK, let me see. I'll adjust the snake AI to prioritize the apple path, and if that's unavailable, rely on a tail-safe path. This approach should enhance decision-making.

**Evaluating options**

I'm considering checking the snake's size versus omitting the safety simulation check. Removing it seems simplest, so I'll search for a fruit path first; otherwise, follow a safe move.

**Strategizing moves**

I’ll direct the snake towards the apple if possible, otherwise, it will fallback to a safe path from its tail. This strategy should efficiently manage the snake's navigation.

**Refining the strategy**

I’m honing the snake AI code to prioritize the apple path, eliminate the safety simulation check, and integrate collision detection with traps and itself to optimize movement and avoid hazards.

**Enhancing snake AI**

I’m refining the snake AI by setting grid parameters, defining movement directions, and drafting functions for BFS pathfinding and collision detection. This improves the game's navigational logic.

**Navigating AI decisions**

The AI prioritizes the apple, seeks the shortest path, or opts for a safe move, ensuring cautious progress. This approach enhances the game strategy and player experience.

**Navigating obstacles**

I’m mapping out the AI's response to the snake's movements, including collision handling, sound effects, and screen updating. This ensures a dynamic and engaging gameplay experience.

Below is an updated version of the code. In this revision the AI always tries to follow a direct path to the fruit when one exists—even if that path may be somewhat “risky.” (In our game, “risky” means that if the move is not possible, the snake will die by colliding with its own body.) When no direct path is found, the AI falls back to following its tail (or any safe move). This change removes the overly conservative simulation check that was causing the snake to “hide” rather than actively chase the apple.

**Note:** Since the game rules disallow moving into any cell currently occupied by any part of the snake (including its tail), the BFS search always treats every snake cell as an obstacle.

Below is the complete code:

import pygame

import random

import sys

from collections import deque

# === Configuration constants ===

CELL\_SIZE = 20

GRID\_WIDTH = 20

GRID\_HEIGHT = 20

WINDOW\_WIDTH = CELL\_SIZE \* GRID\_WIDTH

WINDOW\_HEIGHT = CELL\_SIZE \* GRID\_HEIGHT

FPS = 10 # Frames per second

# Colors (R, G, B)

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

GREEN = (0, 255, 0)

RED = (255, 0, 0)

PURPLE = (128, 0, 128)

# Directions (dx, dy)

UP = (0, -1)

DOWN = (0, 1)

LEFT = (-1, 0)

RIGHT = (1, 0)

# === Helper functions ===

def get\_random\_free\_position(occupied):

"""

Returns a random grid cell (x, y) that is not in the occupied set.

"""

while True:

pos = (random.randint(0, GRID\_WIDTH - 1), random.randint(0, GRID\_HEIGHT - 1))

if pos not in occupied:

return pos

def bfs(start, target, snake, traps):

"""

Uses Breadth-First Search (BFS) to find a path from start to target.

The snake’s body (all segments, including tail) and traps are considered obstacles.

Returns:

A list of grid positions (cells) representing the shortest path from start

to target (excluding the start cell). Returns None if no path is found.

"""

obstacles = set(snake) | set(traps)

queue = deque([start])

came\_from = {start: None}

while queue:

current = queue.popleft()

if current == target:

# Reconstruct path from target back to start.

path = []

while current != start:

path.append(current)

current = came\_from[current]

path.reverse()

return path

# Check all four neighboring cells.

for d in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (current[0] + d[0], current[1] + d[1])

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in came\_from and

next\_cell not in obstacles):

came\_from[next\_cell] = current

queue.append(next\_cell)

return None

def get\_direction(from\_cell, to\_cell):

"""Return the (dx, dy) direction from from\_cell to an adjacent to\_cell."""

return (to\_cell[0] - from\_cell[0], to\_cell[1] - from\_cell[1])

def draw\_grid(surface):

"""Draw grid lines on the surface (for visual effect)."""

for x in range(0, WINDOW\_WIDTH, CELL\_SIZE):

pygame.draw.line(surface, WHITE, (x, 0), (x, WINDOW\_HEIGHT))

for y in range(0, WINDOW\_HEIGHT, CELL\_SIZE):

pygame.draw.line(surface, WHITE, (0, y), (WINDOW\_WIDTH, y))

# === Main game loop ===

def main():

pygame.init()

pygame.mixer.init()

try:

chirp\_sound = pygame.mixer.Sound("chirp.wav")

except pygame.error:

print("Could not load chirp.wav. Please ensure it is in the same folder as this script.")

chirp\_sound = None

screen = pygame.display.set\_mode((WINDOW\_WIDTH, WINDOW\_HEIGHT))

pygame.display.set\_caption("AI Snake Game with Traps")

clock = pygame.time.Clock()

# Initialize snake: starting with 3 segments.

snake = [

(GRID\_WIDTH // 2, GRID\_HEIGHT // 2),

(GRID\_WIDTH // 2 - 1, GRID\_HEIGHT // 2),

(GRID\_WIDTH // 2 - 2, GRID\_HEIGHT // 2)

]

direction = RIGHT

# Initialize traps list (each trap is a grid cell that remains on the board)

traps = []

last\_trap\_time = pygame.time.get\_ticks()

# Place the first fruit (avoid snake and traps)

occupied\_for\_fruit = set(snake) | set(traps)

fruit = get\_random\_free\_position(occupied\_for\_fruit)

running = True

while running:

clock.tick(FPS)

current\_time = pygame.time.get\_ticks()

# Add a new trap every 1 second.

if current\_time - last\_trap\_time >= 1000:

occupied\_for\_trap = set(snake) | set(traps) | {fruit}

new\_trap = get\_random\_free\_position(occupied\_for\_trap)

traps.append(new\_trap)

last\_trap\_time = current\_time

# Process events.

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

# ===== AI DECISION MAKING =====

# Prioritize following a direct path to the fruit.

path\_to\_fruit = bfs(snake[0], fruit, snake, traps)

if path\_to\_fruit is not None:

next\_cell = path\_to\_fruit[0]

direction = get\_direction(snake[0], next\_cell)

else:

# If no path to fruit exists, try following the tail.

path\_to\_tail = bfs(snake[0], snake[-1], snake, traps)

if path\_to\_tail is not None:

next\_cell = path\_to\_tail[0]

direction = get\_direction(snake[0], next\_cell)

else:

# As a last resort, pick any valid move.

for d in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (snake[0][0] + d[0], snake[0][1] + d[1])

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in snake and

next\_cell not in traps):

direction = d

break

# ===== MOVE THE SNAKE =====

new\_head = (snake[0][0] + direction[0], snake[0][1] + direction[1])

# Collision detection: if the new head is outside bounds or in the snake's body, the game ends.

if (new\_head[0] < 0 or new\_head[0] >= GRID\_WIDTH or

new\_head[1] < 0 or new\_head[1] >= GRID\_HEIGHT or

new\_head in snake):

print("Game over! Final score:", len(snake))

running = False

continue

# If the new head lands on a trap, cut the snake's length to half.

if new\_head in traps:

snake.insert(0, new\_head)

new\_length = max(1, len(snake) // 2)

snake = snake[:new\_length]

print("Hit trap! Snake length cut to half. New length:", len(snake))

# If the snake eats the fruit:

elif new\_head == fruit:

snake.insert(0, new\_head)

if chirp\_sound:

chirp\_sound.play()

# Place a new fruit (avoid snake and traps).

occupied\_for\_fruit = set(snake) | set(traps)

fruit = get\_random\_free\_position(occupied\_for\_fruit)

else:

# Normal move: add the new head and remove the tail.

snake.insert(0, new\_head)

snake.pop()

# ===== DRAWING =====

screen.fill(BLACK)

# Draw the fruit.

pygame.draw.rect(screen, RED, (fruit[0] \* CELL\_SIZE, fruit[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Draw the traps.

for trap in traps:

pygame.draw.rect(screen, PURPLE, (trap[0] \* CELL\_SIZE, trap[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Draw the snake.

for segment in snake:

pygame.draw.rect(screen, GREEN, (segment[0] \* CELL\_SIZE, segment[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Optionally, draw the grid lines.

draw\_grid(screen)

pygame.display.update()

pygame.quit()

sys.exit()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Explanation of the Fixes**

1. **Prioritizing the Apple:**
   * The AI now first searches for a path to the fruit using BFS (with the snake’s body and traps as obstacles).
   * If such a path exists, the snake immediately takes the first step along that path.
   * (The earlier version performed a simulation “safety check” that often led the snake to avoid the apple in favor of “safe” moves.)
2. **Fallback Moves:**
   * If no direct path to the fruit is found, the AI then tries to find a path toward the tail.
   * If even that isn’t available, it picks any move that does not cause an immediate collision.
3. **Game-Over on Tail Collision:**
   * The collision check now treats any collision with any part of the snake—including the tail—as game over.
4. **Chirp Sound on Apple Consumption:**
   * When the snake eats the fruit, the code plays the chirp sound (provided that a valid chirp.wav file is present).

This version should cause the snake to more aggressively pursue the apple while still ending the game if a collision (including with its own tail) occurs. Enjoy tweaking and playing your self‑playing snake game!

**Prompt**

*Add a scoring system that rewards better as difficulty increases. So more rewards when the length of the snake is longer and there are more traps. You can also use random colors for apples for variety just do not overlap colors with other objects in the game that are not apples. Use the crash.wav file to play 2-3x consecutively when the snake hits itself or a trap*

**Reasoned about snake game scoring and audio for 2m 47s**

* **So a mixed objective and longer input context takes much longer**

**Advancing the scoring**

I’m thinking through a scoring system that dynamically increases with snake length and traps. Longer snakes and more traps yield higher points. This approach balances progression and challenge.

**Clarifying collision consequences**

OK, let me see. Clarifying if the snake's crash sound means game over or just losing length. Clear instructions are needed for precise implementation.

**Revisiting game rules**

OK, I’m rethinking if the crash sound should signal a game over instead of just reducing snake length when hitting a trap. This might simplify rules and enhance clarity.

**Reassessing collision rules**

Interestingly enough, the updated rules state game over happens if the snake hits itself. Hitting a trap just decreases its length.

**Interpreting the rules**

I’m thinking the instructions mean to play the crash.wav sound 2-3 times on game over and trap events, aligning with the snake's collision responses.

**Developing the scoring system**

I’m setting up a scoring system that rewards progress, boosts the snake's length, and adds more traps. Each apple consumed increases the score, keeping things interesting.

**Crafting the scoring**

OK, let me see. The score could start at 10, then grow with snake length and traps, adding challenge as progress accelerates.

**Piecing together the formula**

I’m crafting a scoring model, where eating an apple boosts the score by 10, plus length of the snake and traps, keeping the challenge interesting.

**Choosing apple colors**

I'm aiming for random apple colors that stay clear of overlaps with the snake, traps, grid, and background colors, adding variety to gameplay.

**Mapping apple colors**

I'm curious about generating random apple colors that harmonize with the game's visual elements, ensuring they don't clash with the snake, traps, grid, or background colors.

**Choosing potential hues**

I’m exploring possible apple colors like orange, yellow, blue, hot pink, and cyan, ensuring they contrast well with the snake, traps, grid, and background. This keeps the game visually engaging.

**Selecting apple hues**

I’m choosing apple colors like orange, yellow, blue, hot pink, and cyan, ensuring they pop against the snake, traps, grid, and background colors for a vibrant blend.

**Piecing together the sound**

I'm considering crash.wav for self-hits or pitfall encounters, replaying it 2-3 times using crash\_sound.play() in a loop for continuous sound.

**Organizing crash sound**

OK, let's arrange to play crash.wav 2-3 times when the snake hits a trap or itself. I'm thinking about for i in range(random.randint(2,3)): crash\_sound.play(); pygame.time.wait(300).

**Ensuring sound clarity**

Including the crash sound in both scenarios seems vital, even if it’s game over for the snake. This approach consistently meets the instructions.

**Toggle mechanisms**

Hm, I'm thinking about using boolean variables to manage the crash sound: crashing\_head can start the sound, and remove\_snake can stop it, softer\_exit softens the sound until the snake exits.

**Handling collisions**

I'm focusing on collision logic for the snake: redressing these involves: checking for collisions, managing snake length and crashes, ensuring continuous sound playback, and preparing for the next iteration.

**Crafting the scoring**

I'm mapping out a scoring system for the snake game, adding points for each apple eaten, with rewards scales based on snake length and traps, and overlaying this on the screen.

**Choosing apple colors**

Mapping out apple hues, avoiding conflicts with snake, trap, grid, and background. Options include red, orange, yellow, blue, and magenta. Progress is being made!

**Selecting apple hues**

I’m refining the list of apple colors, ensuring they don't clash with the snake, trap, grid, or background. The score will be updated and displayed each frame.

Create a score display with Pygame to show progress and manage sound playback. Position the score text at (10,10) and ensure the crash sound loops appropriately.

**Configuring sound responses**

I’m setting up a crash sound that triggers 2-3 times when the snake crashes into itself or a trap, with a 300 ms interval between plays.

**Setting up sound triggers**

I’m mapping out the crash sound to occur 2-3 times for both snake collisions and trap encounters, with a 300 ms interval between plays. Adjust as needed to prevent blocking.

**Integrating sound and scoring**

I’m working on both snake and trap collisions by adding sound. For scoring, I’m incorporating difficulty-based rewards for apple consumption. Progress is clear with this implementation.

**Updating game strategy**

I’m refining snake navigation by incorporating traps and intelligent decision-making with BFS for fruit paths, ensuring a dynamic and engaging gameplay experience.

**Assessing collision outcomes**

Interestingly enough, I’m thinking through the game's collision outcomes, where hitting the snake's tail or traps leads to a crash sound, a score check, and potential game over.

**Integrating sound and snake dynamics**

I’m focusing on snake-trap collisions, ensuring a crash sound aligns with the snake's length halving, and refining the collision detection for a smoother gaming experience.

**Assessing implications**

I'm considering that triggering "crash.wav" during trap collisions might inadvertently affect self-collision sound logic. I'll ensure clarity with the programming team to avoid confusion.

**Ensuring consistent audio**

OK, I’m thinking through the game audio logic to keep "crash.wav" playing with traps and self-collisions. This clarity avoids confusion and enhances the soundtrack's coherence.

**Coordinating collision mechanics**

OK, let me see: the snake's self-collision leads to a game over with a crash sound, while a trap collision only reduces the snake's length, still with the same crash sound playing.

**Separating collision logic**

I’m beginning to see that self-collision and trap collision need to be distinctly handled: self-collision triggers game over, while trap collision shortens the snake's length, both with crash sound notifications.

**Crafting collision handling**

I'm pulling together logic for various events: self-collisions or out-of-bounds triggers a game over, while hitting traps halves the snake's length and continuously plays a crash sound.

import pygame

import random

import sys

from collections import deque

# === Configuration Constants ===

CELL\_SIZE = 20

GRID\_WIDTH = 20

GRID\_HEIGHT = 20

WINDOW\_WIDTH = CELL\_SIZE \* GRID\_WIDTH

WINDOW\_HEIGHT = CELL\_SIZE \* GRID\_HEIGHT

FPS = 10 # Frames per second

# Colors (R, G, B)

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

GREEN = (0, 255, 0) # Snake color

PURPLE = (128, 0, 128) # Trap color

# Candidate apple colors (avoid GREEN, PURPLE, WHITE, BLACK)

APPLE\_COLORS = [

(255, 0, 0), # Red

(255, 165, 0), # Orange

(255, 255, 0), # Yellow

(0, 0, 255), # Blue

(0, 255, 255), # Cyan

(255, 0, 255) # Magenta

]

# Directions (dx, dy)

UP = (0, -1)

DOWN = (0, 1)

LEFT = (-1, 0)

RIGHT = (1, 0)

# === Helper Functions ===

def get\_random\_free\_position(occupied):

"""

Return a random grid cell (x, y) that is not in the occupied set.

"""

while True:

pos = (random.randint(0, GRID\_WIDTH - 1), random.randint(0, GRID\_HEIGHT - 1))

if pos not in occupied:

return pos

def bfs(start, target, snake, traps):

"""

Uses Breadth-First Search (BFS) to find a path from start to target.

The snake’s body (all segments, including the tail) and traps are considered obstacles.

Returns:

A list of grid positions (cells) representing the shortest path from start

to target (excluding the start cell). Returns None if no path is found.

"""

obstacles = set(snake) | set(traps)

queue = deque([start])

came\_from = {start: None}

while queue:

current = queue.popleft()

if current == target:

# Reconstruct the path from target back to start.

path = []

while current != start:

path.append(current)

current = came\_from[current]

path.reverse()

return path

for d in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (current[0] + d[0], current[1] + d[1])

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in came\_from and

next\_cell not in obstacles):

came\_from[next\_cell] = current

queue.append(next\_cell)

return None

def get\_direction(from\_cell, to\_cell):

"""

Return the (dx, dy) direction from from\_cell to an adjacent to\_cell.

"""

return (to\_cell[0] - from\_cell[0], to\_cell[1] - from\_cell[1])

def draw\_grid(surface):

"""

Draw grid lines on the surface for visual effect.

"""

for x in range(0, WINDOW\_WIDTH, CELL\_SIZE):

pygame.draw.line(surface, WHITE, (x, 0), (x, WINDOW\_HEIGHT))

for y in range(0, WINDOW\_HEIGHT, CELL\_SIZE):

pygame.draw.line(surface, WHITE, (0, y), (WINDOW\_WIDTH, y))

def play\_crash\_sound(crash\_sound):

"""

Plays the crash sound 2-3 times consecutively with a short delay.

"""

if crash\_sound:

times = random.randint(2, 3)

for \_ in range(times):

crash\_sound.play()

# Delay to allow the sound to finish (adjust as needed)

pygame.time.wait(300)

# === Main Game Loop ===

def main():

pygame.init()

pygame.mixer.init()

# Load sounds.

try:

chirp\_sound = pygame.mixer.Sound("chirp.wav")

except pygame.error:

print("Could not load chirp.wav. Please ensure it is in the same folder as this script.")

chirp\_sound = None

try:

crash\_sound = pygame.mixer.Sound("crash.wav")

except pygame.error:

print("Could not load crash.wav. Please ensure it is in the same folder as this script.")

crash\_sound = None

screen = pygame.display.set\_mode((WINDOW\_WIDTH, WINDOW\_HEIGHT))

pygame.display.set\_caption("AI Snake Game with Traps, Scoring & Random Apple Colors")

clock = pygame.time.Clock()

# Initialize score and font.

score = 0

font = pygame.font.SysFont("Arial", 24)

# Initialize snake: starting with 3 segments.

snake = [

(GRID\_WIDTH // 2, GRID\_HEIGHT // 2),

(GRID\_WIDTH // 2 - 1, GRID\_HEIGHT // 2),

(GRID\_WIDTH // 2 - 2, GRID\_HEIGHT // 2)

]

direction = RIGHT

# Initialize traps list (each trap is a grid cell that remains on the board)

traps = []

last\_trap\_time = pygame.time.get\_ticks()

# Place the first apple (avoid snake and traps)

occupied\_for\_fruit = set(snake) | set(traps)

fruit\_pos = get\_random\_free\_position(occupied\_for\_fruit)

fruit\_color = random.choice(APPLE\_COLORS)

running = True

while running:

clock.tick(FPS)

current\_time = pygame.time.get\_ticks()

# Add a new trap every 1 second.

if current\_time - last\_trap\_time >= 1000:

occupied\_for\_trap = set(snake) | set(traps) | {fruit\_pos}

new\_trap = get\_random\_free\_position(occupied\_for\_trap)

traps.append(new\_trap)

last\_trap\_time = current\_time

# Process events.

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

# ===== AI DECISION MAKING =====

# Prioritize following a direct path to the apple.

path\_to\_fruit = bfs(snake[0], fruit\_pos, snake, traps)

if path\_to\_fruit is not None:

next\_cell = path\_to\_fruit[0]

direction = get\_direction(snake[0], next\_cell)

else:

# If no direct path exists, try following the tail.

path\_to\_tail = bfs(snake[0], snake[-1], snake, traps)

if path\_to\_tail is not None:

next\_cell = path\_to\_tail[0]

direction = get\_direction(snake[0], next\_cell)

else:

# As a last resort, choose any valid move.

for d in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (snake[0][0] + d[0], snake[0][1] + d[1])

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in snake and

next\_cell not in traps):

direction = d

break

# ===== MOVE THE SNAKE =====

new\_head = (snake[0][0] + direction[0], snake[0][1] + direction[1])

# Check for collisions with walls or the snake's body (including tail).

if (new\_head[0] < 0 or new\_head[0] >= GRID\_WIDTH or

new\_head[1] < 0 or new\_head[1] >= GRID\_HEIGHT or

new\_head in snake):

play\_crash\_sound(crash\_sound)

print("Game over! Final score:", score)

running = False

continue

# If the snake hits a trap, play crash sound, then cut its length to half.

elif new\_head in traps:

play\_crash\_sound(crash\_sound)

snake.insert(0, new\_head)

new\_length = max(1, len(snake) // 2)

snake = snake[:new\_length]

print("Hit trap! Snake length cut to half. New length:", len(snake))

# If the snake eats the apple.

elif new\_head == fruit\_pos:

snake.insert(0, new\_head)

if chirp\_sound:

chirp\_sound.play()

# Increase score based on snake length and number of traps.

# (The harder it is, the higher the reward.)

apple\_reward = 10 + len(snake) + len(traps)

score += apple\_reward

print("Apple eaten! Score increased by", apple\_reward, "New score:", score)

# Place a new apple (avoid snake and traps), with a random color.

occupied\_for\_fruit = set(snake) | set(traps)

fruit\_pos = get\_random\_free\_position(occupied\_for\_fruit)

fruit\_color = random.choice(APPLE\_COLORS)

else:

# Normal move.

snake.insert(0, new\_head)

snake.pop()

# ===== DRAWING =====

screen.fill(BLACK)

# Draw the apple.

pygame.draw.rect(screen, fruit\_color, (fruit\_pos[0] \* CELL\_SIZE, fruit\_pos[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Draw the traps.

for trap in traps:

pygame.draw.rect(screen, PURPLE, (trap[0] \* CELL\_SIZE, trap[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Draw the snake.

for segment in snake:

pygame.draw.rect(screen, GREEN, (segment[0] \* CELL\_SIZE, segment[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Draw grid lines.

draw\_grid(screen)

# Draw the score.

score\_text = font.render("Score: " + str(score), True, WHITE)

screen.blit(score\_text, (10, 10))

pygame.display.update()

pygame.quit()

sys.exit()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Explanation of Key Changes**

* **Scoring:**  
  A score variable is initialized at 0. Each time an apple is eaten, the score increases by

*apple\_reward = 10 + len(snake) + len(traps)*

This means that as the snake grows longer and more traps appear, the reward per apple increases. The score is rendered on the screen using a Pygame font.

* **Random Apple Colors:**  
  When placing an apple, the code randomly picks a color from the list APPLE\_COLORS. This gives the apples variety while ensuring the colors don’t conflict with the snake (GREEN), traps (PURPLE), grid (WHITE), or background (BLACK).
* **Crash Sound Effects:**  
  A helper function play\_crash\_sound() plays the crash.wav sound 2–3 times consecutively with a short delay (300 ms) between plays. This function is called when the snake collides with itself (game over) or when it hits a trap (which cuts its length to half).

Enjoy this enhanced version of the self‑playing Snake game that now rewards riskier play and features more audio/visual variety!