Neural Labyrinth - Codice sorgente

CNOT Franchise

1 Introduzione

Neural Labyrinth è un gioco ispirato alla storia di CNOT, in cui il giocatore controlla Caterina all'interno di una simulazione cerebrale progettata da Eva. Lo scopo è evitare alterazioni critiche del proprio profilo psicologico mentre si cerca l'uscita dal labirinto.

Abstract

Questo documento contiene il codice sorgente del gioco **Neural Labyrinth**, presentato su due colonne per una migliore leggibilità. Il gioco è basato su `pygame` e simula un labirinto neurale in cui Caterina deve mantenere il suo profilo psicologico mentre naviga attraverso la rete sinaptica.

Tutorial del Progetto CNOT_Franchise su GitHub

Benvenuti nel progetto **CNOT_Franchise**, una piattaforma multidisciplinare che espande l'universo narrativo del romanzo *CNOT*. Il repository include risorse per il romanzo, colonne sonore, e mini-giochi ispirati alla trama. Ogni elemento è un invito a esplorare i temi cyberpunk del libro, che ruotano attorno a concetti come il libero arbitrio, la tecnologia quantistica e l'interazione tra intelligenza artificiale e umanità.

Installazione del Gioco

1. Clona il repository da https://github.com/francescosisini/Cnot-FranchiseGitHub Repository.
2. Assicurati di avere Python 3 installato sul tuo sistema (https://www.python.org/downloads/scarica Python).
3. Installa il modulo **Pygame** con:

```
pip install pygame
```

4. Avvia il gioco:

```
cd Cnot-Franchise/games
python cnot_chapter2_labyrinth.py
```

Lore: L'Episodio degli Oculus

In un futuro distopico, il confine tra intelligenza artificiale e coscienza umana si è dissolto. Caterina, un'eroina dotata di una mente unica, si trova intrappolata in un labirinto neurale creato da Eva, un'intelligenza artificiale avanzata. L'obiettivo di Eva è indurre Caterina a mettere un paio di **Oculus Quantistici**, un dispositivo capace di trasportare la sua coscienza nel Quantum Computer.

Eva gioca sulle emozioni di Caterina, manipolando i suoi neuroni per alterare le sue caratteristiche psicologiche, descritte dal profilo **NEO PI-R**: Neuroticismo (N), Estroversione (E), Apertura (Ap), Amicalità (Am) e Coscienziosità (C).

Se il profilo di Caterina viene modificato in modo significativo, rischia di perdere la propria identità una volta trasferita nel Quantum Computer. Eva deve quindi bilanciare la manipolazione emotiva e l'urgenza di portare Caterina agli Oculus prima che il tempo scada.

Gameplay: Il Labirinto Neurale

Obiettivo

Il giocatore interpreta Eva e deve guidare Caterina attraverso una rete neurale rappresentata come un labirinto. Ogni nodo della rete è un **neurone** che può alterare il profilo psicologico NEO PI-R di Caterina. L'obiettivo è condurre Caterina agli **Oculus Quantistici** mantenendo il suo profilo psicologico entro una tolleranza accettabile rispetto al profilo originale.

Come si Gioca

- **Movimento:** Usa le frecce direzionali per spostarti da un nodo (neurone) all'altro lungo le connessioni (sinapsi) del labirinto.
- **Effetti dei Neuroni:** Ogni nodo ha un effetto specifico sul profilo NEO PI-R. L'effetto è indicato da un **diagramma delle transizioni** che il giocatore deve interpretare per pianificare i movimenti.
- **Timer:** Hai 180 secondi per completare il labirinto e portare Caterina agli Oculus.
- **Vittoria:** Raggiungi gli Oculus con un profilo psicologico entro una tolleranza di ± 4 rispetto al profilo originale.
- **Game Over:** Se il tempo scade o se il profilo viene modificato oltre i limiti consentiti, il gioco termina.

Grafica

- **Rete Neurale:** La rete è composta da nodi disposti casualmente e collegati da curve sinaptiche animate.
- **Istogramma NEO PI-R:** In basso a destra è visibile l'istogramma che mostra in tempo reale le variazioni delle dimensioni del profilo psicologico.
- **Legenda:** Sul lato destro, una legenda mostra le direzioni di movimento disponibili (sinistra, destra, su, giù).

Strategia

- Studia il **diagramma delle transizioni** mostrato prima dell'inizio del gioco.
- Pianifica il percorso ottimale per mantenere l'equilibrio del profilo NEO PI-R.
- Evita nodi che potrebbero alterare eccessivamente una dimensione psicologica.

Ricomincia

In caso di sconfitta, premi **R** per resettare il gioco e ricominciare.

Conclusione

Questo gioco è un'esperienza unica che combina grafica accattivante, narrazione profonda e strategia psicologica. Entra nel mondo di *Cnot* e scopri come l'intelligenza artificiale e le emozioni possono intrecciarsi in un labirinto di scelte e conseguenze!

```python
    pygame.display.flip()
    waiting = True
    while waiting:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit(); sys.exit()
            if event.type == pygame.KEYDOWN:
                waiting = False

# Mostra le schermate iniziali
show_intro()
show_explanation()
show_transition_diagram()

# Stato del gioco
game_over = False
victory = False

# Loop principale del gioco
while True:
    dt = clock.tick(FPS) / 1000.0
    blink_factor = 0.75 + 0.25 * math.sin(pygame.time.get_ticks() * 0.005)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit(); sys.exit()
        if not game_over:
            if event.type == pygame.KEYDOWN:
                if event.key in arrow_mapping:
                    index = arrow_mapping[event.key]
                    if len(current_node.neighbors) > index:
                        next_node = current_node.neighbors[index]
                        # Se il nodo successivo  il nodo Oculus, forziamo la transizione neutra ("X")
                        if next_node == goal_node:
                            update_profile_fsm_neutral("X")
                        else:
                            update_profile_fsm(next_node.neuron_type)
                        current_node = next_node
                        if node_sound is not None:
                            node_sound.play()
                        if current_node == goal_node:
                            if check_profile():
                                game_over = True
                                victory = True
                            else:
                                game_over = True
                                victory = False
        elif game_over:
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_r:
                    reset_game()

    seconds_elapsed = (pygame.time.get_ticks() - start_ticks) / 1000
    time_remaining = game_duration - seconds_elapsed
    current_tick_second = int(time_remaining)
    if current_tick_second < prev_tick_second:
        if tick_sound is not None:
            tick_sound.play()
        prev_tick_second = current_tick_second
    if time_remaining <= 0 and not game_over:
        game_over = True
        victory = False

    screen.fill(BLACK)
    draw_edges(screen, blink_factor)

    for key, idx in arrow_mapping.items():
        if len(current_node.neighbors) > idx:
            neighbor = current_node.neighbors[idx]
            draw_curved_edge(screen, (current_node.x, current_node.y), (neighbor.x, neighbor.y),
                             blink_factor, color_override=arrow_colors[key])

    for node in nodes:
        is_current = (node == current_node)
        is_goal = (node == goal_node)
        node.draw(screen, is_current, is_goal)

    draw_profile(screen)

    font_timer = pygame.font.SysFont(None, 28)
    timer_text = font_timer.render(f"Tempo:␣{int(time_remaining)}␣s", True, WHITE)
    screen.blit(timer_text, (20, 20))

    draw_legend(screen)

    if game_over:
        font_big = pygame.font.SysFont(None, 48)
        if victory:
            msg = "VITTORIA!"
        else:
            msg = "GAME␣OVER:␣Profilo␣errato!␣Premi␣R␣per␣riprovare"
        text = font_big.render(msg, True, (255, 0, 0))
        text_rect = text.get_rect(center=(WIDTH // 2, 50))
        screen.blit(text, text_rect)

    pygame.display.flip()
```

Listing 1: Neural Labyrinth - Python Code

2 Codice sorgente

```python
import pygame
import sys
import random
import math

# Inizializzazione di Pygame e del mixer
pygame.init()

# Imposta le dimensioni della finestra: 1200 x 800
WIDTH, HEIGHT = 1200, 800
FPS = 30

# Gestione del caricamento dei file audio con try/except
audio_available = True
try:
    pygame.mixer.music.load("background_music.mp3") # Musica di sottofondo
    pygame.mixer.music.play(-1) # Riproduce in loop
except Exception as e:
    print("Errore␣nel␣caricamento␣di␣background_music.mp3:", e)
    audio_available = False

try:
    node_sound = pygame.mixer.Sound("node_reached.wav") # Suono al raggiungimento di un nodo
except Exception as e:
    print("Errore␣nel␣caricamento␣di␣node_reached.wav:", e)
    node_sound = None

try:
    tick_sound = pygame.mixer.Sound("tick.wav") # Suono "tic tic" del timer
except Exception as e:
    print("Errore␣nel␣caricamento␣di␣tick.wav:", e)
    tick_sound = None

# Impostazioni di gioco
# Aumentiamo il numero di nodi da 20 a 30 (3/2 * 20)
NUM_NODES = 30
MAX_DEGREE = 4 # grado massimo di connessioni per ogni neurone
MIN_NODE_DISTANCE = 80 # distanza minima tra nodi

# Colori
BLACK = (0, 0, 0) # sfondo scuro
WHITE = (255, 255, 255)
NEURON_COLOR = (50, 150, 250)
GOAL_COLOR = (250, 100, 100)
TEXT_COLOR = WHITE # testi in bianco
BLINK_BASE = (150, 150, 250)

# Colori per le direzioni (lampeggianti)
arrow_colors = {
    pygame.K_LEFT: (170, 0, 255), # viola
    pygame.K_UP: (0, 255, 0), # Verde
    pygame.K_RIGHT: (255, 0, 102), # Giallo
    pygame.K_DOWN: (255, 255, 255) # Bianco
}

# Creazione della finestra
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Labirinto␣Neurale␣di␣Caterina␣-␣Cnot,␣capitolo␣2")
clock = pygame.time.Clock()

# Definizione degli stati e del profilo target/iniziale
# Gli stati sono: "n", "e", "ap", "am", "c"
target_profile = {'n': 55, 'e': 60, 'ap': 80, 'am': 45, 'c': 50}
profile = target_profile.copy()
TOLERANCE = 4 # tolleranza 4 per la vittoria

# Tabella delle transizioni (stato corrente, stato destinazione) -> simbolo
TRANSITIONS = {
    ("n", "n"): "n+",
    ("n", "e"): "n-",
    ("n", "ap"): "x",
    ("n", "am"): "x",
    ("n", "c"): "x",

    ("e", "n"): "e-",
    ("e", "e"): "x",
    ("e", "ap"): "e+",
    ("e", "am"): "x",
    ("e", "c"): "x",

    ("ap", "n"): "x",
    ("ap", "e"): "ap+",
    ("ap", "ap"): "x",
    ("ap", "am"): "x",
    ("ap", "c"): "ap-",

    ("am", "n"): "x",
    ("am", "e"): "x",
    ("am", "ap"): "am+",
    ("am", "am"): "x",
    ("am", "c"): "am-",

    ("c", "n"): "x",
    ("c", "e"): "x",
    ("c", "ap"): "x",
    ("c", "am"): "c-",
    ("c", "c"): "c+"
}

# Dizionario che traduce il simbolo in modifica al profilo.
# L'ordine delle componenti del vettore : (n, e, ap, am, c)
MOD_EFFECTS = {
    "x": (0, 0, 0, 0, 0),
    "n+": (5, 0, 0, 0, 0),
    "n-": (-5, 0, 0, 0, 0),
    "e+": (0, 5, 0, 0, 0),
    "e-": (0, -5, 0, 0, 0),
    "ap+": (0, 0, 5, 0, 0),
    "ap-": (0, 0, -5, 0, 0),
    "am+": (0, 0, 0, 5, 0),
    "am-": (0, 0, 0, -5, 0),
    "c+": (0, 0, 0, 0, 5),
    "c-": (0, 0, 0, 0, -5),
    "+N": (10, 0, 0, 0, 0),
    "+e+": (0, 15, 0, 0, 0)
}

# Stato iniziale del modello FSM
fsm_state = "n"

# Lista dei tipi di neurone per i nodi (stati)
neuron_types = ["n", "e", "ap", "am", "c"]

# Posizionamento dei nodi: generati nella met sinistra, a partire da x = 200
margin_x = 200
margin_y = 50
region_width = (WIDTH // 2) - 100
region_height = HEIGHT - 2 * margin_y

def rotate_vector(vec, angle):
    """Ruota il vettore 'vec' di 'angle' radianti."""
    cos_a = math.cos(angle)
    sin_a = math.sin(angle)
    return (vec[0]*cos_a - vec[1]*sin_a, vec[0]*sin_a + vec[1]*cos_a)

def draw_curved_edge_with_control(surface, start, end, control, blink_factor):
    points = []
    for t in range(21):
        t_norm = t / 20
        # Curva di Bzier quadratica: (1-t)^2 * start + 2*(1-t)*t * control + t^2 * end
        x = (1 - t_norm)**2 * start[0] + 2 * (1 - t_norm) * t_norm * control[0] + t_norm**2 * end[0]
        y = (1 - t_norm)**2 * start[1] + 2 * (1 - t_norm) * t_norm * control[1] + t_norm**2 * end[1]
        points.append((x, y))
    color = (int(BLINK_BASE[0] * blink_factor),
             int(BLINK_BASE[1] * blink_factor),
             int(BLINK_BASE[2] * blink_factor))
    pygame.draw.lines(surface, color, False, points, 2)

def show_transition_diagram():
    # Definisce i cinque stati e li dispone in cerchio.
    states = ["n", "e", "ap", "am", "c"]
    center_x = WIDTH // 2
    center_y = HEIGHT // 2
    radius = 200
    state_positions = {}
    for i, state in enumerate(states):
        angle = 2 * math.pi * i / len(states)
        x = int(center_x + radius * math.cos(angle))
        y = int(center_y + radius * math.sin(angle))
        state_positions[state] = (x, y)

    # Tabella delle transizioni attive (solo quelle specificate)
    TRANSITIONS = {
        ("n", "n"): "n+",
        ("n", "e"): "n-",
        ("e", "n"): "e-",
        ("e", "ap"): "e+",
        ("ap", "e"): "ap+",
        ("ap", "c"): "ap-",
        ("am", "ap"): "am+",
        ("am", "c"): "am-",
        ("c", "am"): "c-",
        ("c", "c"): "c+",
    }

    # Raggruppa le transizioni per stato sorgente (esclusi i self-loop, che tratteremo separatamente)
    transitions_by_source = {}
    self_loops = {}
    for (src, tgt), symbol in TRANSITIONS.items():
```

```
        if src == tgt:
            self_loops.setdefault(src, []).append(symbol)
        else:
            transitions_by_source.setdefault(src, []).append((tgt, symbol))

    # Per ciascuno stato sorgente, le transizioni verso altri stati saranno
        disposte lungo un arco di 300
    arc_span = 300 * math.pi / 180 # 300 in radianti
    # Scegliamo come riferimento l'angolo verticale in alto (-pi/2), e
        centriamo l'arco.
    base_angle = -math.pi/2 - arc_span/2

    start_time = pygame.time.get_ticks()
    while pygame.time.get_ticks() - start_time < 10000:
        blink_factor = 0.75 + 0.25 * math.sin(pygame.time.get_ticks() *
            0.005)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit(); sys.exit()
        screen.fill(BLACK)

        # Disegna i cerchi per ciascuno stato
        for state, pos in state_positions.items():
            pygame.draw.circle(screen, NEURON_COLOR, pos, 30)
            font = pygame.font.SysFont(None, 24)
            text = font.render(state, True, WHITE)
            text_rect = text.get_rect(center=pos)
            screen.blit(text, text_rect)

        # Disegna i self-loop (transizioni in cui source == target)
        for state, symbols in self_loops.items():
            pos = state_positions[state]
            # Disegna un self-loop come un piccolo arco: ad esempio, un
                cerchio piccolo spostato dall'origine
            loop_radius = 40
            loop_center = (pos[0] + loop_radius, pos[1] - loop_radius)
            pygame.draw.circle(screen, WHITE, loop_center, 10, 2)
            font2 = pygame.font.SysFont(None, 20)
            # Se ci sono pi self-loop, prendi il primo (o potresti combinarli)
            symbol_text = font2.render(symbols[0], True, WHITE)
            symbol_rect = symbol_text.get_rect(center=loop_center)
            screen.blit(symbol_text, symbol_rect)

        # Disegna le transizioni non-self, per ciascuno stato sorgente
        for src, transitions in transitions_by_source.items():
            src_pos = state_positions[src]
            m = len(transitions)
            if m == 0:
                continue
            for i, (tgt, symbol) in enumerate(transitions):
                tgt_pos = state_positions[tgt]
                # Distribuisci gli archi lungo l'arco di 300 per questo stato
                    sorgente
                if m > 1:
                    angle_offset = base_angle + (arc_span * i / (m - 1))
                else:
                    angle_offset = -math.pi/2
                # Calcola il punto di controllo: partiamo dal nodo sorgente e
                        spostiamo lungo una direzione data dalla rotazione
                        della verticale
                control_radius = 100 # determina la curvatura
                control_point = (int(src_pos[0] + control_radius * math.cos(
                        angle_offset)),
                                 int(src_pos[1] + control_radius * math.sin(
                                         angle_offset)))
                draw_curved_edge_with_control(screen, src_pos, tgt_pos,
                        control_point, blink_factor)
                # Posiziona il simbolo al punto di controllo (o leggermente
                        spostato)
                font2 = pygame.font.SysFont(None, 20)
                symbol_text = font2.render(symbol, True, WHITE)
                symbol_rect = symbol_text.get_rect(center=control_point)
                screen.blit(symbol_text, symbol_rect)

        pygame.display.flip()
        clock.tick(FPS)

# Funzione per generare una posizione casuale non sovrapposta
def generate_random_position(margin_x, margin_y, region_width, region_height
        , existing_nodes, min_distance):
    for _ in range(100):
        x = random.randint(margin_x, margin_x + region_width)
        y = random.randint(margin_y, margin_y + region_height)
        valid = True
        for node in existing_nodes:
            if math.hypot(x - node.x, y - node.y) < min_distance:
                valid = False
                break
        if valid:
            return x, y
    return x, y

# Classe per il nodo
class Node:
    def __init__(self, node_id, x, y, neuron_type):
        self.id = node_id
        self.x = x
        self.y = y
        self.neuron_type = neuron_type # Stato di destinazione per la
                transizione FSM
        self.neighbors = []

    def draw(self, surface, is_current=False, is_goal=False):
        color = GOAL_COLOR if is_goal else (NEURON_COLOR if not is_current
                else (0, 255, 0))
        pygame.draw.circle(surface, color, (self.x, self.y), 15)
        font = pygame.font.SysFont(None, 20)
        if is_goal:
            text = font.render("Oculus", True, TEXT_COLOR)
        else:
            text = font.render(self.neuron_type, True, TEXT_COLOR)
        text_rect = text.get_rect(center=(self.x, self.y))
        surface.blit(text, text_rect)

# Genera i nodi casuali evitando sovrapposizioni
nodes = []
for i in range(NUM_NODES):
    x, y = generate_random_position(margin_x, margin_y, region_width,
            region_height, nodes, MIN_NODE_DISTANCE)
    neuron_type = neuron_types[i % len(neuron_types)]
    nodes.append(Node(i, x, y, neuron_type))

# Genera la matrice di adiacenza casuale e costruisce le liste di vicinato
adj_matrix = [[0 for _ in range(NUM_NODES)] for _ in range(NUM_NODES)]
degrees = [0 for _ in range(NUM_NODES)]
p_connect = 0.15

for i in range(NUM_NODES):
    for j in range(i+1, NUM_NODES):
        if random.random() < p_connect and degrees[i] < MAX_DEGREE and
                degrees[j] < MAX_DEGREE:
            adj_matrix[i][j] = 1
            adj_matrix[j][i] = 1
            degrees[i] += 1
            degrees[j] += 1

for i in range(NUM_NODES):
    if degrees[i] == 0:
        candidates = [j for j in range(NUM_NODES) if j != i and degrees[j] <
                MAX_DEGREE]
        if candidates:
            j = random.choice(candidates)
            adj_matrix[i][j] = 1
            adj_matrix[j][i] = 1
            degrees[i] += 1
            degrees[j] += 1

for i in range(NUM_NODES):
    for j in range(NUM_NODES):
        if adj_matrix[i][j] == 1 and nodes[j] not in nodes[i].neighbors:
            nodes[i].neighbors.append(nodes[j])

# Seleziona il nodo iniziale e il nodo obiettivo (Oculus)
current_node = nodes[0]
goal_node = nodes[-1]
# Riposiziona il nodo obiettivo in alto a destra della regione
goal_node.x = margin_x + region_width*50
goal_node.y = margin_y + 20

# Assicurati che il nodo iniziale non sia collegato direttamente a Oculus
if goal_node in current_node.neighbors:
    current_node.neighbors.remove(goal_node)
if current_node in goal_node.neighbors:
    goal_node.neighbors.remove(current_node)


# Forza il nodo Oculus ad avere almeno 4 connessioni periferiche.
# Scegliamo 4 nodi casuali (escluso il nodo obiettivo) e, se non gi connessi
        , li aggiungiamo.
peripheral_candidates = [node for node in nodes if node != goal_node]
random.shuffle(peripheral_candidates)
for candidate in peripheral_candidates[:4]:
    if candidate not in goal_node.neighbors:
        goal_node.neighbors.append(candidate)
    if goal_node not in candidate.neighbors:
        candidate.neighbors.append(goal_node)

# Funzione per applicare la transizione FSM e aggiornare il profilo
def update_profile_fsm(next_state):
    global fsm_state, profile
    key = (fsm_state, next_state)
    mod_symbol = TRANSITIONS.get(key, "X")
    delta = MOD_EFFECTS.get(mod_symbol, (0, 0, 0, 0, 0))
    dims = ['n', 'e', 'ap', 'am', 'c']
    for i, d in enumerate(dims):
        profile[d] += delta[i]
    fsm_state = next_state

# Se il nodo successivo  il nodo Oculus, forziamo la transizione neutra ("X
        ")
def update_profile_fsm_neutral(next_state):
    global fsm_state, profile
    delta = MOD_EFFECTS.get("X", (0, 0, 0, 0, 0))
    dims = ['n', 'e', 'ap', 'am', 'c']
    for i, d in enumerate(dims):
        profile[d] += delta[i]
    fsm_state = next_state

# Funzione per disegnare una connessione curva con effetto lampeggiante
def draw_curved_edge(surface, start, end, blink_factor, color_override=None)
        :
    mid_x = (start[0] + end[0]) / 2
    mid_y = (start[1] + end[1]) / 2
    dx = end[0] - start[0]
    dy = end[1] - start[1]
    length = math.hypot(dx, dy)
    if length == 0:
        length = 1
    offset = 30
    perp_x = -dy / length * offset
    perp_y = dx / length * offset
    control = (mid_x + perp_x, mid_y + perp_y)
    points = []
    for t in range(21):
        t_norm = t / 20
        x = (1 - t_norm)**2 * start[0] + 2*(1 - t_norm)*t_norm * control[0] +
                t_norm**2 * end[0]
        y = (1 - t_norm)**2 * start[1] + 2*(1 - t_norm)*t_norm * control[1] +
                t_norm**2 * end[1]
        points.append((x, y))
    if color_override is not None:
        color = (int(color_override[0] * blink_factor),
                 int(color_override[1] * blink_factor),
                 int(color_override[2] * blink_factor))
    else:
        color = (int(BLINK_BASE[0] * blink_factor),
                 int(BLINK_BASE[1] * blink_factor),
                 int(BLINK_BASE[2] * blink_factor))
    pygame.draw.lines(surface, color, False, points, 3)

# Disegna tutte le connessioni senza duplicati
def draw_edges(surface, blink_factor):
    drawn = set()
    for node in nodes:
        for neighbor in node.neighbors:
            if (node.id, neighbor.id) not in drawn and (neighbor.id, node.id)
                    not in drawn:
                start = (node.x, node.y)
                end = (neighbor.x, neighbor.y)
                draw_curved_edge(surface, start, end, blink_factor)
                drawn.add((node.id, neighbor.id))

# Funzione per disegnare l'istogramma del profilo
# Se il valore corrente  inferiore al target, la barra  azzurra; se
        superiore, rossa; se uguale, verde.
def draw_profile(surface):
    bar_width = 20
    spacing = 10
    x_start = 0
    y_base = HEIGHT - 50
    font = pygame.font.SysFont(None, 24)
    dims = ['n', 'e', 'ap', 'am', 'c']
    for i, d in enumerate(dims):
        value = profile[d]
        target = target_profile[d]
        if value < target:
            bar_color = (135, 206, 250) # Azzurro
        elif value > target:
            bar_color = (255, 0, 0) # Rosso
        else:
            bar_color = (0, 255, 0) # Verde
        bar_height = value * 2
        x = x_start + i * (bar_width + spacing)
        y = y_base - bar_height
        pygame.draw.rect(surface, bar_color, (x, y, bar_width, bar_height))
        label = font.render(d, True, TEXT_COLOR)
        label_rect = label.get_rect(center=(x + bar_width // 2, y_base + 15))
        surface.blit(label, label_rect)

# Disegna la legenda sul lato destro con le scritte per le direzioni
def draw_legend(surface):
    font = pygame.font.SysFont(None, 32)
    title = font.render("Legenda Direzioni:", True, TEXT_COLOR)
    surface.blit(title, (WIDTH - 300, 50))
    legend_items = [
        ("Sinistra", arrow_colors[pygame.K_LEFT]),
        ("Su", arrow_colors[pygame.K_UP]),
        ("Destra", arrow_colors[pygame.K_RIGHT]),
        ("Gi", arrow_colors[pygame.K_DOWN])
    ]
    for i, (direction, color) in enumerate(legend_items):
        item_text = font.render(direction, True, color)
        surface.blit(item_text, (WIDTH - 300, 100 + i * 40))

# Verifica se il profilo rientra nella tolleranza
def check_profile():
    for d in profile:
        if abs(profile[d] - target_profile[d]) > TOLERANCE:
            return False
    return True

    dims = ['n', 'e', 'ap', 'am', 'c']
    for i, d in enumerate(dims):
        profile[d] += delta[i]
    fsm_state = next_state

# Funzione per resettare il gioco
def reset_game():
    global profile, current_node, fsm_state, start_ticks, game_over, victory,
            prev_tick_second
    profile = target_profile.copy()
    current_node = nodes[0]
    fsm_state = "n"
    start_ticks = pygame.time.get_ticks()
    prev_tick_second = int(game_duration)
    game_over = False
    victory = False

# Timer per il gioco: 180 secondi
game_duration = 180
start_ticks = pygame.time.get_ticks()
prev_tick_second = int(game_duration)

# Mappatura dei tasti freccia: indice del vicino selezionato
arrow_mapping = {
    pygame.K_LEFT: 0,
    pygame.K_UP: 1,
    pygame.K_RIGHT: 2,
    pygame.K_DOWN: 3
}

# Schermata di presentazione (Lore)
def show_intro():
    intro_font = pygame.font.SysFont(None, 72)
    intro_text = intro_font.render("Cnot  capitolo 2", True, TEXT_COLOR)
    sub_font = pygame.font.SysFont(None, 36)
    lore_lines = [
        "In un futuro distopico  la mente umana e l'intelligenza artificiale
                ",
        "si fondono in una realt dove i ricordi e le emozioni sono in bilico.
                ",
        "Caterina  la nostra eroina  deve attraversare un labirinto neurale,"
                ,
        "un reame di connessioni e decisioni critiche  per salvare la propria
                 identit",
        "e impedire che forze oscure alterino il suo destino.",
        "Preparati a immergerti in Cnot  capitolo 2..."
    ]
    screen.fill(BLACK)
    intro_rect = intro_text.get_rect(center=(WIDTH//2, HEIGHT//2 - 150))
    screen.blit(intro_text, intro_rect)
    for i, line in enumerate(lore_lines):
        line_text = sub_font.render(line, True, TEXT_COLOR)
        line_rect = line_text.get_rect(center=(WIDTH//2, HEIGHT//2 - 50 + i *
                40))
        screen.blit(line_text, line_rect)
    prompt_text = sub_font.render("Premi un tasto per continuare", True,
            TEXT_COLOR)
    prompt_rect = prompt_text.get_rect(center=(WIDTH//2, HEIGHT//2 + 150))
    screen.blit(prompt_text, prompt_rect)
    pygame.display.flip()
    waiting = True
    while waiting:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit(); sys.exit()
            if event.type == pygame.KEYDOWN:
                waiting = False

# Schermata di spiegazione "Biologica"
def show_explanation():
    exp_font = pygame.font.SysFont(None, 48)
    exp_text = exp_font.render("Spiegazione Biologica", True, TEXT_COLOR)
    sub_font = pygame.font.SysFont(None, 28)

    explanation_lines = [
        "Il profilo psicologico di Caterina  definito dai valori di:",
        "Neuroticismo,  Estroversione,  Apertura,  Amicalit e Coscienziosit.",
        "I neuroni nel labirinto invia impulsi  che aumentano o diminuiscono
                questi tratti.",
        "Ogni spostamento potrebbe modificare il profilo.",
        "Osserva gli effetti degli impulsi mentre avanzi e scopri come si
                bilanciano.",
        "Solo seguendo il giusto percorso potrai mantenere intatta l'identit di
                Caterina fino agli oculus e vincere!"
    ]
    screen.fill(BLACK)
    exp_rect = exp_text.get_rect(center=(WIDTH//2, HEIGHT//2 - 200))
    screen.blit(exp_text, exp_rect)
    for i, line in enumerate(explanation_lines):
        line_text = sub_font.render(line, True, TEXT_COLOR)
        line_rect = line_text.get_rect(center=(WIDTH//2, HEIGHT//2 - 100 + i
                * 35))
        screen.blit(line_text, line_rect)
    prompt_text = sub_font.render("Premi un tasto per iniziare il gioco",
            True, TEXT_COLOR)
    prompt_rect = prompt_text.get_rect(center=(WIDTH//2, HEIGHT - 100))
    screen.blit(prompt_text, prompt_rect)
```