Automi a stati finiti (ASF o FSM)

DI MASSIMO VAILATI
06/02/2026

PYTHON AUTOMI VIDEOGIOCHI SIMULAZIONE STATI COMPORTAMENTI

LICEO , SCIENTIFICO S.A. , TT INFORMATICA

Contenuto: introduzione agli automi a stati finiti come modelli matematici fondamentali per rappresentare comportamenti sequenziali, validare input e modellare sistemi reattivi, mostrando la loro utilità sia nella didattica che nella progettazione di software e giochi.

Attività pratica: la realizzazione di un automa per validare numeri interi positivi, la simulazione degli stati di un personaggio in un videogioco, e la creazione di un prototipo di videogioco con controllo del personaggio.

Autori

MASSIMO VAILATI

Nel mondo dell'informatica teorica, i linguaggi formali e gli automi a stati finiti (Finite State Machines, FSM) rappresentano strumenti fondamentali per comprendere come funzionano la computazione, la validazione dei dati e persino alcune logiche di gioco. Anche se spesso vengono percepiti come concetti astratti, possono rivelarsi estremamente pratici nella didattica, soprattutto se tradotti in attività laboratoriali concrete.

Gli automi a stati finiti (ASF) sono modelli matematici usati per rappresentare comportamenti sequenziali. Il loro scopo è descrivere un sistema che, in ogni momento, si trova esattamente in uno di un numero finito di stati possibili. Gli ASM sono capaci di:

Rappresentare comportamenti sequenziali (es. stati di un personaggio di gioco: "fermo", "cammina", "salta"...)

Validare stringhe di input (es. controllare se un identificatore è valido, o se una password rispetta un formato)

Modellare sistemi reattivi (es. menù interattivi, sistemi di controllo, semafori)

Semplificare la progettazione e il debugging: uno stato alla volta, nessuna ambiguità

Un automa a stati finiti è composto da:

un insieme finito di stati
- Tutte le possibili condizioni in cui il sistema si può trovare

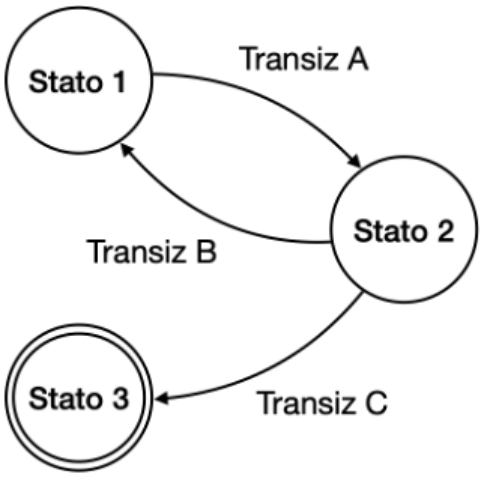
uno stato iniziale
- Lo stato in cui il sistema si trova all'inizio.

uno o più stati finali (a seconda del contesto)
- Sono opzionali e indicano un risultato valido o completato.

un alfabeto di simboli in ingresso
- L'insieme di tutti i possibili eventi o simboli che causano un cambiamento

una funzione di transizione
- Regola che indica come cambiare stato in base all'input

Una possibile rappresentazione grafica è un **diagramma o grafo orientato**: ciascun nodo è uno *stato*, e le frecce etichettate indicano le *transizioni*.



In questo esempio l'ASF ha 3 stati: Stato 1, Stato 2 e Stato 3 (stato finale). La Transizione A porta l'ASF dallo Stato 1 allo Stato 2, la Transizione B porta l'ASF dallo Stato 2 allo Stato1 e la Transizione C porta l'ASF dallo Stato 2 allo Stato 3.

ESEMPI PRATICI DI ASF

1 - Il Semaforo Stradale

Questo è l'esempio classico di un sistema reattivo modellato da un ASF.

Stati: Rosso, Verde, Giallo.

Input (Alfabeto): Un segnale di **Tempo Scaduto** (il timer interno).

Funzione di Transizione:

DA Rosso + Input **Tempo Scaduto A Verde**

DA Verde + Input **Tempo Scaduto A Giallo**

DA Giallo + Input **Tempo Scaduto A Rosso**

In questo modello, lo stato attuale *determina* il prossimo stato, garantendo un ciclo di funzionamento coerente e non ambiguo.

2 – Il distributore automatico

Un distributore automatico è un sistema che deve **ricordare** lo stato corrente dei soldi inseriti per determinare le azioni successive.

Stati

Gli stati rappresentano l'ammontare totale di credito inserito e definiscono ciò che l'utente può fare:

- **S0 (Iniziale): Nessun Credito** (0€). La macchina è in attesa di monete.
- **S1 Credito Parziale** (ad esempio, 0.50€). Non sufficiente per il prodotto.
- **S2 (Finale): Credito Sufficiente** (ad esempio, 1.00€ o più). Il prodotto può essere erogato

Alfabeto di Input

Gli input sono gli eventi che provocano le transizioni:

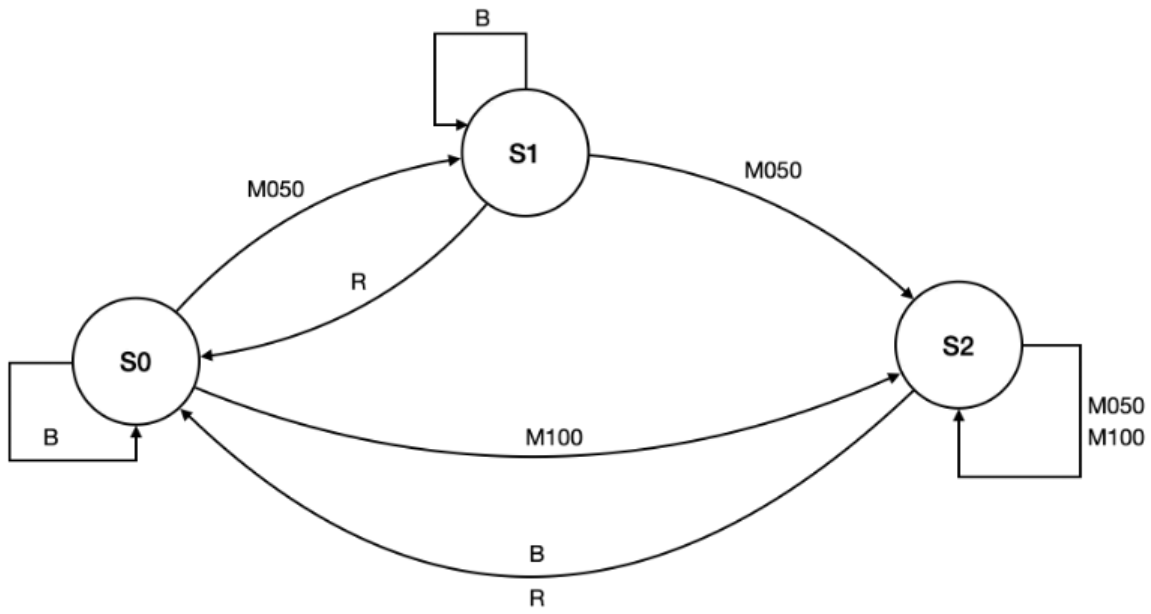
- **M050**: Inserimento di una moneta da 0.50€.
- **M100**: Inserimento di una moneta da 1.00€.
- **B**: Pressione del **Pulsante di Acquisto** (se il credito è sufficiente).
- **R**: Pressione del **Pulsante di Ritiro Resto**.

Funzione di Transizione: Regole di Funzionamento

Le transizioni determinano come si muove il sistema tra gli stati.

Stato Attuale	Input Ricevuto	Stato Successivo	Azione della Macchina (Output)
S0(0.00€)	M050	S1 (0.50€)	Nessuna. Aggiorna il credito.
S0(0.00€)	M100	S2 (1.00€)	Nessuna. Aggiorna il credito.
S1 (0.50€)	M050	S2 (1.00€)	Nessuna. Aggiorna il credito.
S1 (0.50€)	M100	S2 (1.50€)	Nessuna. Aggiorna il credito.
S2 (>=1.00€)	M050	S2 (Credito +0.50€)	Nessuna. Aggiorna il credito.
S2 (>=1.00€)	M100	S2 (Credito +1.00€)	Nessuno. Aggiorna il credito
S2 (>=1.00€)	B (Acquisto)	S0 (0.00€)	Eroga il prodotto; Eroga il resto.
S2 / S1	R (Resto)	S0 (0.00€)	Eroga tutto il credito come resto.
S0 / S1	B (Acquisto)	Stato Corrente	Lampeggia LED: "Credito Insufficiente!" (Output di errore)

Rappresentazione mediante grafo



ATTIVITÀ PRATICA N.1

Validazione dell'input

La validazione dell'input è uno dei contesti più efficaci per far comprendere agli studenti il potere dei linguaggi formali e degli automi a stati finiti. Molti programmatori usano subito le espressioni regolari per controllare il formato di un dato, ma spesso non capiscono davvero *perché* funzionano o *che cosa* rappresentano.

Usare una FSM permette invece di far emergere:

- la logica formale dietro la validazione
- il concetto di riconoscimento di linguaggi
- l'utilità pratica degli automi nella progettazione di software robusto

Implementiamo un automa a stati finiti che **riconosca numeri interi positivi**, ovvero stringhe costituite da una o più cifre (`0`–`9`) con:

- nessuno spazio
- nessuna lettera
- nessun simbolo
- almeno una cifra

Esempi accettati:`"0","5","123","999999"`
Esempi rifiutati:`"","  123","12 34","-5","12.5","3a7"`

Progetto dell'automa

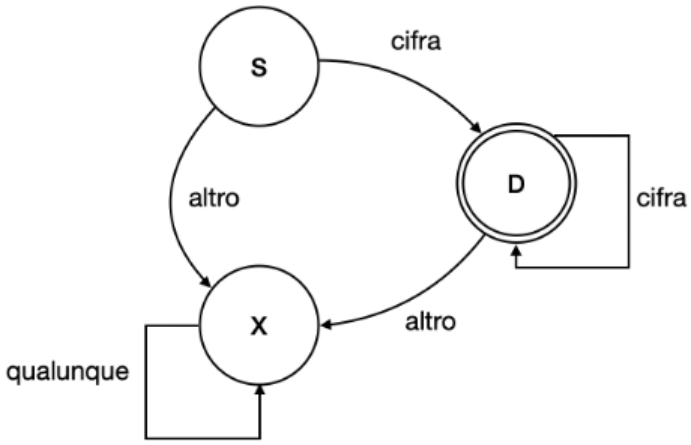
Stati

- **S** – stato iniziale
  Attende la prima cifra
- **D** – "dentro numero"
  Dopo aver letto almeno una cifra
- **X** – stato di errore
  Se arriva un carattere non valido
- **D è stato accettante**

Transizioni

Stato	Input	Nuovo stato
S	cifra	D
S	altro	X
D	cifra	D
D	altro	X
X	qualunque	X

Grafo



Codice Python
Ecco una versione della FSM completamente implementata (*input.py*)

```
# Insieme degli stati dell'automa:
# S = stato iniziale
# D = stato "dentro numero" (accettante)
# X = stato di errore
states = {"S", "D", "X"}

# Stato iniziale
start = "S"

# Stati accettanti (qui solo D)
accepting = {"D"}

def get_symbol(c):
    # Restituisce il tipo di simbolo letto.
    # L'automa non lavora direttamente sui caratteri,
    # ma su categorie astratte: 'digit' o 'other'.
    return "digit" if c.isdigit() else "other"

# Funzione di transizione:
# (stato_corrente, simbolo) → nuovo_stato
transitions = {
    ("S", "digit"): "D",    # La prima cifra rende il numero valido
    ("S", "other"): "X",    # Se già il primo carattere non è cifra → errore

    ("D", "digit"): "D",    # Altre cifre mantengono la validità
    ("D", "other"): "X",    # Qualunque carattere non numerico dopo cifre → errore

    ("X", "digit"): "X",    # Una volta in errore, si rimane in errore
    ("X", "other"): "X",
}

def run_fsm(s):
    # Esegue l'automa sulla stringa s.
    # Ritorna True se s viene riconosciuta come numero intero positivo.

    # Lo stato corrente parte sempre dallo stato iniziale
    state = start

    # Analizza la stringa carattere per carattere
    for c in s:
```

```
        # Converte il carattere in un 'simbolo' astratto
        symbol = get_symbol(c)

        # Applica la transizione definita nell'automa
        state = transitions[(state, symbol)]

    # La stringa è accettata solo se lo stato finale è uno stato accettante
    return state in accepting

# -------------------------
# TEST DI VALIDAZIONE
# -------------------------

print(run_fsm("12345"))  # True: tutte cifre
print(run_fsm("12a45"))  # False: contiene una lettera
print(run_fsm(""))       # False: stringa vuota → rimane in S (non accettante)
print(run_fsm("0"))      # True: singola cifra è valida
print(run_fsm(" 123"))   # False: spazio iniziale → errore
print(run_fsm("42 "))    # False: spazio finale → errore
```

ATTIVITÀ PRATICA N.2

Controllare il personaggio di un videogioco

Immaginiamo un semplice videogioco 2D di tipo platform. Il personaggio può:

- stare fermo (**Idle**)
- camminare (**Walk**)
- saltare (**Jump**)
- subire un danno (**Hit**)
- essere KO (**KO**)

Ogni stato definisce **cosa il personaggio sta facendo e quali transizioni sono possibili**.

Stati possibili e logica

1. Idle
Il personaggio non sta compiendo azioni.
Transizioni:

- Idle → Walk (se viene premuto un tasto di movimento)
- Idle → Jump (se si preme il tasto “salta”)
- Idle → Hit (se si subisce un attacco)

2. Walk
Il personaggio si muove orizzontalmente.
Transizioni:

- Walk → Idle (se si rilasciano i tasti di movimento)
- Walk → Jump (se si preme “salta”)
- Walk → Hit (se colpito)

3. Jump
Il personaggio è in aria.

Transizioni:

- Jump → Idle (quando atterra)
- Jump → Hit (se viene colpito in aria)

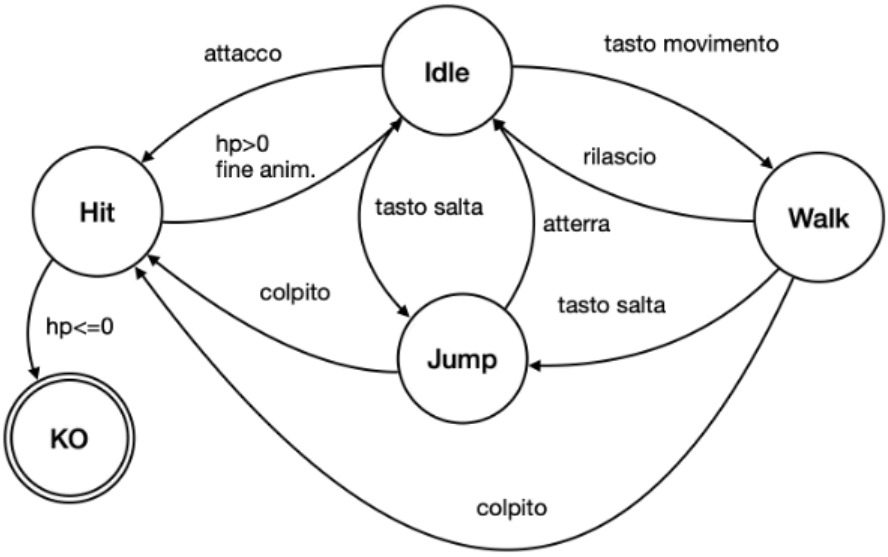
4. Hit
Il personaggio è colpito e riproduce un'animazione di danno.
Transizioni:

- Hit → Idle (se resta HealthPoints > 0 e finisce l'animazione)
- Hit → KO (se HealthPoints <= 0)

5. KO
Il personaggio è sconfitto. Stato finale.
Transizioni:

- KO → (nessuna)

Grafo



Codice Python

Il codice seguente (*character.py*) propone una struttura semplice da far implementare agli studenti.
Non è un motore di gioco completo, ma una base concettuale.

```
class CharacterFSM:
    def __init__(self):
        self.state = "Idle"
        self.hp = 3

    def handle_event(self, event):
        print(f"Stato attuale: {self.state}. Evento: {event}")

        if self.state == "Idle":
            if event == "move":
                self.state = "Walk"
            elif event == "jump":
                self.state = "Jump"
```

```
            elif event == "hit":
                self.state = "Hit"

        elif self.state == "Walk":
            if event == "stop":
                self.state = "Idle"
            elif event == "jump":
                self.state = "Jump"
            elif event == "hit":
                self.state = "Hit"

        elif self.state == "Jump":
            if event == "land":
                self.state = "Idle"
            elif event == "hit":
                self.state = "Hit"

        elif self.state == "Hit":
            self.hp -= 1
            if self.hp <= 0:
                self.state = "KO"
            else:
                self.state = "Idle"

        print(f"Nuovo stato: {self.state}\n")

char = CharacterFSM()
char.handle_event("move")
char.handle_event("jump")
char.handle_event("land")
char.handle_event("hit")
char.handle_event("hit")
char.handle_event("hit")
```

Esecuzione

```
Stato attuale: Idle. Evento: move
Nuovo stato: Walk

Stato attuale: Walk. Evento: jump
Nuovo stato: Jump

Stato attuale: Jump. Evento: land
Nuovo stato: Idle

Stato attuale: Idle. Evento: hit
Nuovo stato: Hit

Stato attuale: Hit. Evento: hit
Nuovo stato: Idle

Stato attuale: Idle. Evento: hit
Nuovo stato: Hit
```

ATTIVITÀ PRATICA N.3

Prototipo di videogioco con controllo del personaggio

Il codice seguente (*gameplay.py*) propone la realizzazione di un prototipo di videogioco con personaggio controllato dallo stesso FSM considerato nell'attività pratica numero 2. Per l'esecuzione è necessario installare pygame (`pip install pygame`).

Comportamenti implementati:

Stati: `Idle, Walk, Jump, Hit, KO`
Tasti:
- Frecce sinistra/destra → movimento
- Barra spaziatrice → salto

Gravità e atterraggio simulati
Colpo che riduce HP e può mandare KO

Il programma visualizza un personaggio come un semplice rettangolo che cambia colore in base allo stato.

Idle (grigio) → fermo
Walk (blu) → si muove
Jump (giallo) → salta
Hit (rosso) → colpito
KO (rosso scuro) → sconfitto

Il log della console mostra tutte le transizioni.

Per rendere dinamico il gioco è stato aggiunto

Un **nemico** rappresentato da un quadrato rosso scuro che si muove avanti e indietro.
Una funzione di **collisione** tra personaggio ed enemy.
Quando c'è collisione → la FSM del personaggio riceve l'evento `"hit"`.

Codice Python

```
import pygame
import sys

# -----------------------
#     FSM DEL PERSONAGGIO
# -----------------------
class CharacterFSM:
    def __init__(self):
        # Stato iniziale del personaggio
        self.state = "Idle"
        # Punti vita
        self.hp = 10
        # Velocità verticale usata per la fisica del salto
        self.y_velocity = 0
        # Flag che indica se il personaggio è a terra (può saltare)
        self.is_on_ground = True

    def handle_event(self, event):
        # Stampa evento e stato corrente (utile per debug)
        print(f"EVENTO: {event}, Stato: {self.state}")

        # LOGICA DEGLI STATI: transizioni basate sull'evento ricevuto
        if self.state == "Idle":
```

```
            if event == "move":
                # Quando ci si muove da fermi -> camminare
                self.state = "Walk"
            elif event == "jump":
                # Inizio salto
                self.state = "Jump"
            elif event == "hit":
                # Subito dopo essere colpiti
                self.state = "Hit"

        elif self.state == "Walk":
            if event == "stop":
                # Stop movimento -> tornare fermi
                self.state = "Idle"
            elif event == "jump":
                # Saltare anche mentre si cammina
                self.state = "Jump"
            elif event == "hit":
                # Colpito mentre cammina
                self.state = "Hit"

        elif self.state == "Jump":
            if event == "land":
                # Atterraggio -> tornare a Idle
                self.state = "Idle"
            elif event == "hit":
                # Colpito in aria
                self.state = "Hit"

        elif self.state == "Hit":
            # Quando si entra nello stato Hit si decrementano gli HP
            self.hp -= 1
            if self.hp <= 0:
                # Se gli HP finiscono -> KO permanente
                self.state = "KO"
            else:
                # Altrimenti si torna a Idle dopo l'animazione di hit
                self.state = "Idle"

        # Stampa il nuovo stato (utile per debug)
        print(f" → Nuovo stato: {self.state}")

# ------------------------
#     SETUP DI GIOCO
# ------------------------
pygame.init()
screen = pygame.display.set_mode((800, 600))
clock = pygame.time.Clock()

# Posizione iniziale del personaggio (x, y)
x = 400
y = 500
# Velocità orizzontale del personaggio
speed = 5
# Accelerazione dovuta alla gravità usata per il salto
gravity = 0.8
# Istanza della macchina a stati del personaggio
fsm = CharacterFSM()
```

```
# ----------------------
#     NEMICO
# ----------------------
# Posizione del nemico
enemy_x = 200
enemy_y = 500
# Velocità di movimento del nemico e direzione (1 destra, -1 sinistra)
enemy_speed = 3
enemy_direction = 1  # 1 = destra, -1 = sinistra

# ----------------------
#   LOOP PRINCIPALE
# ----------------------
while True:
    # Lettura input tastiera
    keys = pygame.key.get_pressed()
    moved = False  # flag per capire se il personaggio si è mosso orizzontalmente

    # Gestione eventi di sistema (es. chiusura finestra)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # --- MOVIMENTO ORIZZONTALE ---
    if keys[pygame.K_LEFT]:
        x -= speed
        moved = True
    if keys[pygame.K_RIGHT]:
        x += speed
        moved = True

    # Transizioni tra Idle <-> Walk in base al movimento orizzontale
    if moved and fsm.state == "Idle":
        fsm.handle_event("move")
    if not moved and fsm.state == "Walk":
        fsm.handle_event("stop")

    # --- SALTO ---
    # Se premi SPACE e sei a terra, inizia il salto
    if keys[pygame.K_SPACE] and fsm.is_on_ground:
        fsm.is_on_ground = False
        # Impulso iniziale verso l'alto (valore negativo per y decresce verso l'alto nello schermo)
        fsm.y_velocity = -15
        fsm.handle_event("jump")

    # ----------------------------
    #   FISICA DEL SALTO
    # ----------------------------
    if not fsm.is_on_ground:
        # Aggiorno posizione verticale in base alla velocità verticale
        y += fsm.y_velocity
        # Applico gravità incrementando la velocità verso il basso
        fsm.y_velocity += gravity

        # Rilevo atterraggio sul "terreno" (y = 500)
        if y >= 500:
            y = 500
```

```
        fsm.is_on_ground = True
        fsm.y_velocity = 0
        # Al ritorno a terra, se ero in Jump genero evento land
        if fsm.state == "Jump":
            fsm.handle_event("land")

# ----------------------------
#   MOVIMENTO DEL NEMICO
# ----------------------------
# Sposta il nemico e inverte direzione ai limiti
enemy_x += enemy_speed * enemy_direction

if enemy_x <= 100 or enemy_x >= 650:  # limiti di pattuglia del nemico
    enemy_direction *= -1

# ----------------------------
#   COLLISIONE PERSONAGGIO / NEMICO
# ----------------------------
# Rette di collisione (possono essere sostituite da hitbox più precise)
player_rect = pygame.Rect(x, y, 50, 80)
enemy_rect = pygame.Rect(enemy_x, enemy_y, 50, 80)

# Se c'è collisione e il personaggio non è già KO, riceve un colpo
if player_rect.colliderect(enemy_rect) and fsm.state != "KO":
    fsm.handle_event("hit")

# ----------------------------
#   RENDER
# ----------------------------
# Pulisco lo schermo
screen.fill((30, 30, 30))

# Mappa colori per visualizzare lo stato corrente del personaggio
colors = {
    "Idle": (200, 200, 200),  # Grigio chiaro
    "Walk": (50, 150, 255),   # Blu
    "Jump": (255, 200, 50),   # Giallo
    "Hit": (255, 80, 80),     # Rosso chiaro
    "KO": (80, 0, 0)          # Rosso scuro
}

# Disegno del personaggio con colore in base allo stato
pygame.draw.rect(screen, colors[fsm.state], player_rect)

# Disegno del nemico
pygame.draw.rect(screen, (180, 0, 0), enemy_rect)

# Testo informativo in alto a sinistra (stato e HP)
font = pygame.font.SysFont(None, 32)
text = font.render(f"Stato: {fsm.state}   HP: {fsm.hp}", True, (255, 255, 255))
screen.blit(text, (10, 10))

# Aggiorno schermo e limito il frame rate
pygame.display.flip()
clock.tick(60)
```