

LUCA AMIRANTE

Software Developer

# Game development con Pygame

Develer webinar

30 Nov 2022

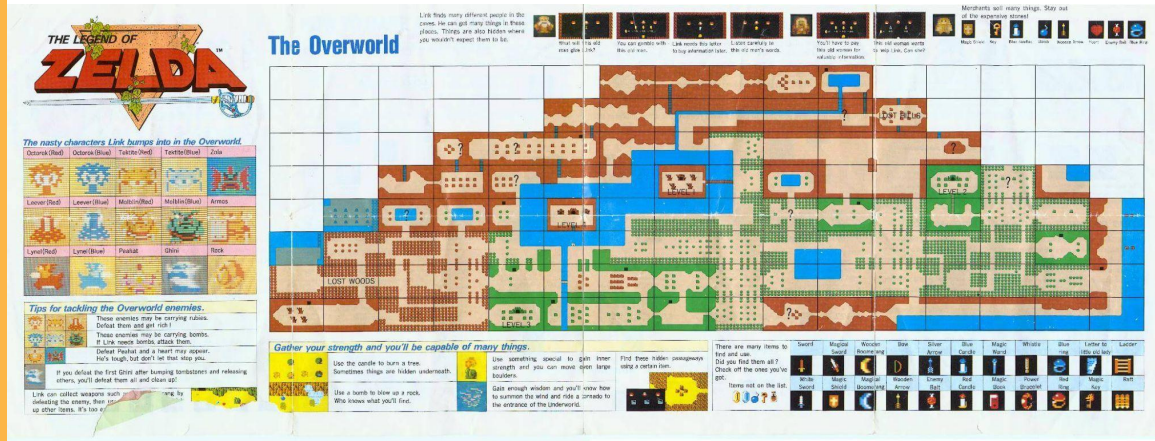
develer

# I COSA TRATTEREMO

A partire dall'idea di creare un gioco in stile ***The Legend of Zelda***, impareremo:

- I Concetti di game design
- I Concetti di game development
- I Sviluppare videogiochi con Pygame

# IL PROCESSO DI SVILUPPO



“Quando ero bambino, sono andato in montagna e mi sono imbattuto in un lago. Fu una sorpresa scoparlo. Viaggiando per il paese senza una mappa, tentando di trovare la mia strada, imbattendomi in cose stupefacenti lungo il cammino, realizzai come ci si sente a vivere tali avventure.”

Shigeru Miyamoto

Il processo di sviluppo

## CORE CONCEPT

- È l'idea su cui il gioco si basa.
- Deve essere una frase concisa, che racchiuda l'essenza del gioco.

## Core concept – Il processo di sviluppo

*“Un gioco in cui il protagonista esplora un mondo sconosciuto, dove incontra diversi tipi di nemici e prove da affrontare.”*



# CORE MECHANICS

- Sono le azioni che il giocatore ripeterà più e più volte durante il gioco.
- Definiscono il genere del gioco.
- Definiscono quanto il gioco sia divertente.
- Possono cambiare in fase di prototipazione.

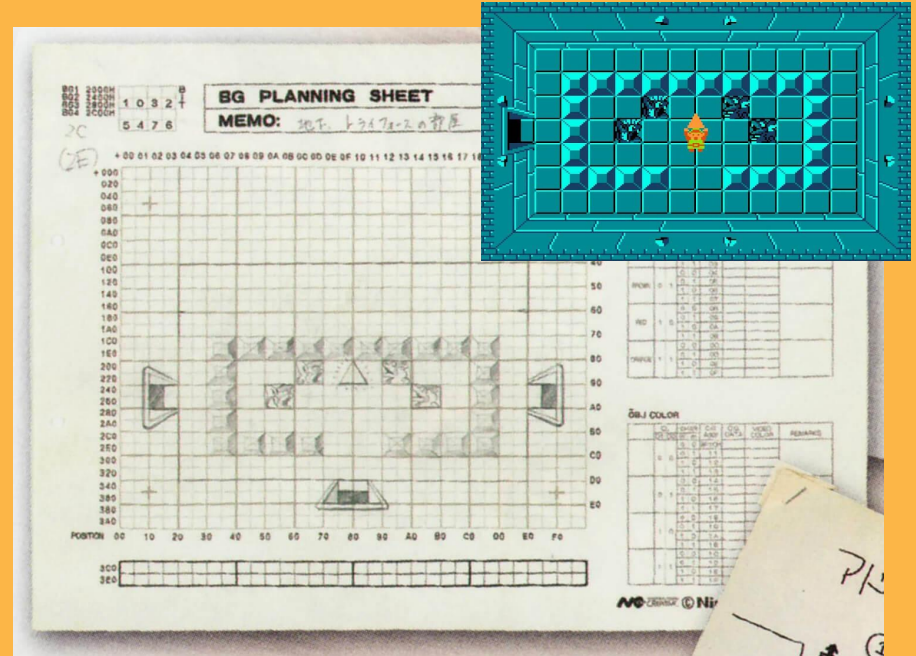
## Core mechanics - Il processo di sviluppo

- Vista dall'alto.
- Utilizzo della spada per attaccare i nemici.
- Esplorazione.
- Sono le caratteristiche di un *action-adventure*.



# IL PROTOTIPO – Il processo di sviluppo

Siamo pronti a  
sviluppare un prototipo!





Il processo di sviluppo

# IL PROTOTIPO

## PERCHÉ

- Testare le meccaniche principali di gioco.
- Valutare se il *gameplay* è divertente/interessante.

Il processo di sviluppo

# IL PROTOTIPO COME

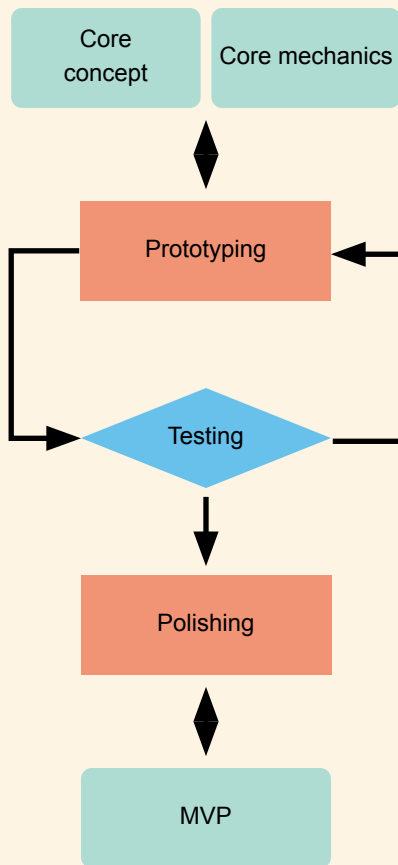
- | Sviluppato nel minor tempo possibile.
- | Solo meccaniche principali.
- | Non utilizza assets (o ne utilizza di temporanei).

Il processo di sviluppo

## Roadmap

Realizza un prototipo e testalo finché non ti soddisfa.

Arriva a un prodotto minimo funzionante (MVP).



Definisci i concetti e le meccaniche.

Aggiungi contenuto e rifinisci le meccaniche.

# INTRODUZIONE A PYGAME

Cos'è Pygame e  
che vantaggi offre



# COS'È PYGAME

È un insieme di moduli Python progettati per la scrittura di videogiochi, che si basa sulla libreria [SDL](#) e ne amplia le funzionalità.

*Vedi la sezione [About](#) della documentazione.*

# PERCHÉ PYGAME

- | Caratteristiche positive di Python.
- | Ottimo strumento educativo.

# LIMITI DI PYGAME

- Limiti di *Python* e *SDL*.
- Alcuni dei limiti si possono aggirare.

Per approfondire:



Pygame's Performance -  
What You Need to Know

117.243 visualizzazioni • 1 anno fa



DaFluffyPotato

## I **PASSIAMO AL CODICE!** – Introduzione a Pygame

Pygame si installa molto facilmente tramite il comando:

```
pip install pygame
```



## I Inizializziamo Pygame

Per inizializzare `pygame`,  
chiamiamo `pygame.init()`.

```
import pygame as pg

class Game:
    """Classe principale di gioco."""

    def __init__(self):
        # Inizializza i moduli di pygame.
        pg.init()

if __name__ == "__main__":
    # Crea un'istanza di `Game`,
    # avviando, così, il gioco.
    Game()
```

## I Creiamo una finestra

Inizializziamo la finestra.

Vedi [display](#).

```
TITOLO = "Game development con Pygame"
```

```
RISOLUZIONE = (800, 450)
```

```
class Game:
```

```
    def __init__(self):
```

```
        pg.init()
```

```
        self.screen = self.init_screen()
```

```
    def init_screen(self) -> pg.Surface:
```

```
        # Setta il nome della finestra di gioco.
```

```
        pg.display.set_caption(TITOLO)
```

```
        # Inizializza la finestra, settandone
```

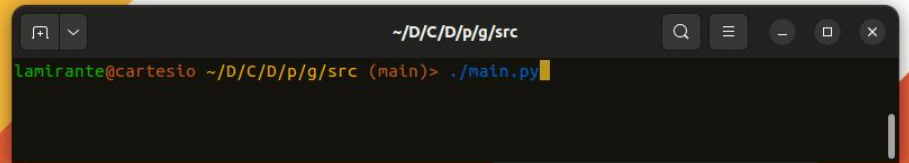
```
        # la risoluzione.
```

```
        return pg.display.set_mode(RISOLUZIONE)
```

# I CREIAMO UNA FINESTRA – Introduzione a Pygame

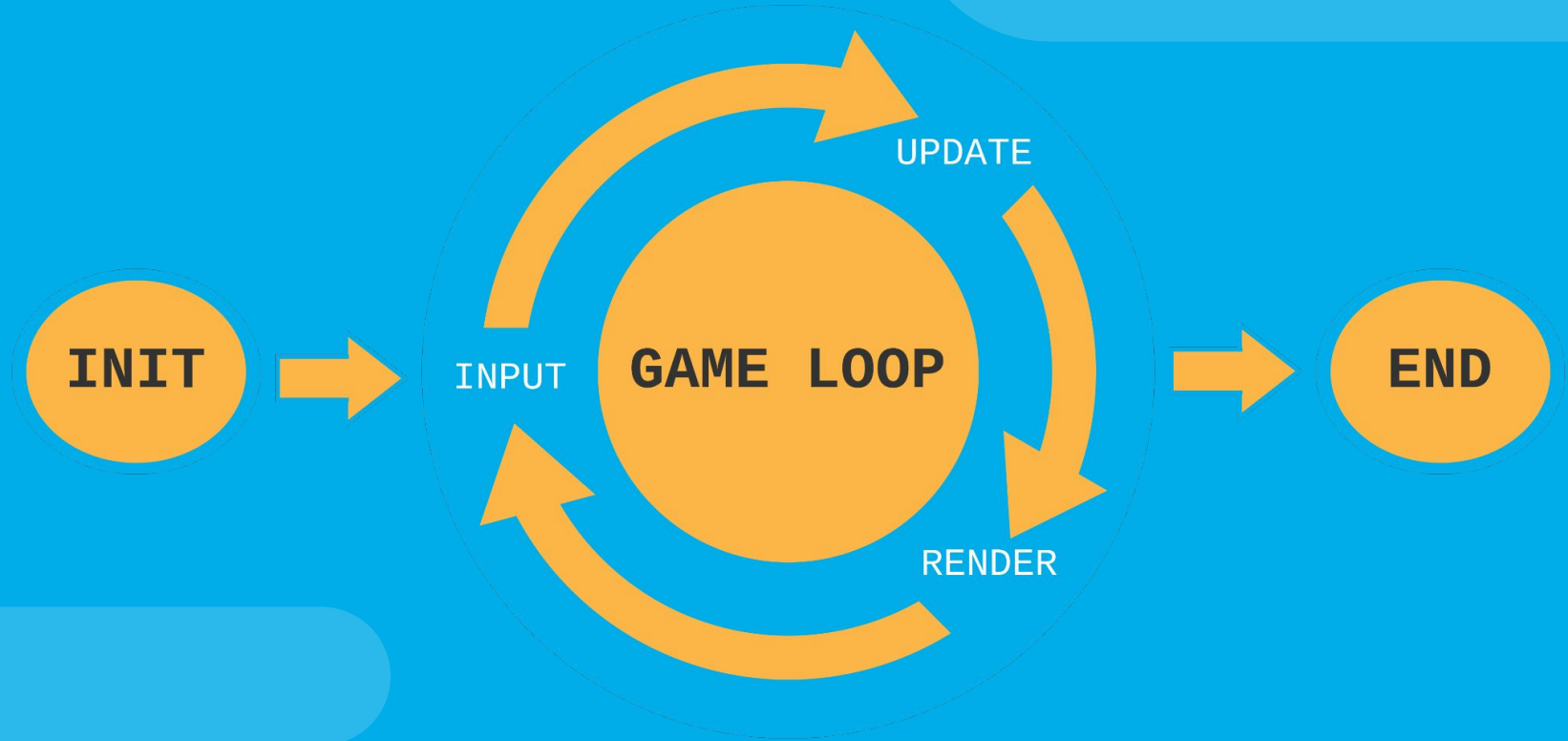
Una volta inizializzata la finestra,  
l'esecuzione arriva al termine.

È il momento di introdurre il  
*game loop*!

A terminal window with a dark background. The title bar shows the path ~/D/C/D/p/g/src. The prompt is lamirante@cartesio ~/D/C/D/p/g/src (main)>. The command ./main.py has been entered and is highlighted with a yellow cursor.

```
~/D/C/D/p/g/src  
lamirante@cartesio ~/D/C/D/p/g/src (main)> ./main.py
```

## I IL FLUSSO DEL GIOCO



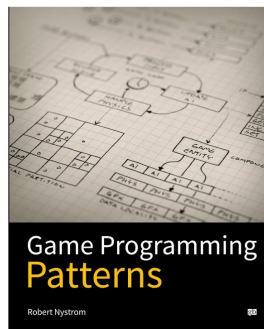
# INIZIALIZZAZIONE

- Carica le impostazioni.
- Carica gli assets.
- Inizializza la finestra di gioco.
- Inizializza il *game loop*.
- Eseguita una sola volta all'avvio del gioco.

# GAME LOOP

- È il corpo del gioco.
- Contiene i processi che vanno ripetuti continuamente per tutta la durata dell'esecuzione.

Per approfondire:



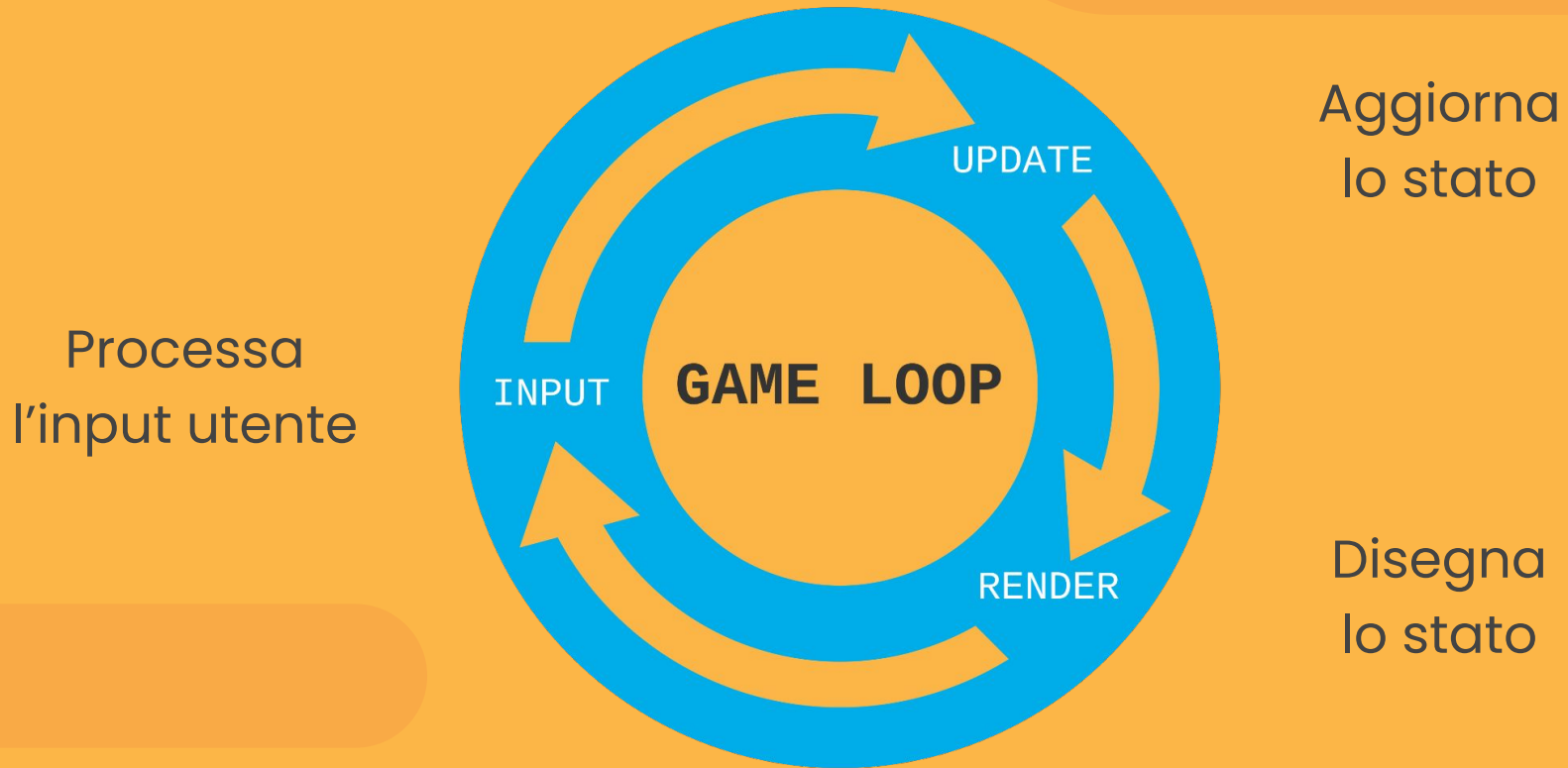
## Game Loop

[Game Programming Patterns](#) / [Sequencing Patterns](#)

# FINALIZZAZIONE

- | Eseguita al termine del *game loop*.
- | Termina tutti i processi.
- | Salva i dati utente, se necessario.

## II GAME LOOP – Il game loop





## I Implementazione - Il game loop

```
class Game:

    def __init__(self): ...
        self.run_game_loop() # Avvia il game loop.

    def run_game_loop(self):
        while True:
            self.process_events() # In Pygame, l'input è gestito tramite eventi.
            self.update()
            self.draw() # In Pygame, draw è il metodo standard per indicare il rendering.
```

## I Gestione input - Il game loop

Pygame gestisce l'input (e non solo) tramite eventi.

Vedi [pygame.event](#).

```
class Game:

    def process_events(self):
        # Rimuove gli eventi presenti nella queue di Pygame
        # e li restituisce sotto forma di una lista.
        for event in pg.event.get():

            # catturiamo l'evento QUIT, che rappresenta il click sulla x della finestra.
            if event.type == pg.QUIT:
                sys.exit()
```

## Eventi

- Emessi tramite azioni specifiche.
- Gli attributi ci danno maggiori informazioni.
- È possibile definire eventi personalizzati.

Vedi [event](#).

# Evento	# Attributi
QUIT	none
KEYDOWN	key, mod, unicode, scancode
KEYUP	key, mod, unicode, scancode
MOUSEMOTION	pos, rel, buttons, touch
MOUSEBUTTONUP	pos, button, touch
MOUSEBUTTONDOWN	pos, button, touch
JOYAXISMOTION	instance_id, axis, value
JOYBALLMOTION	instance_id, ball, rel
JOYHATMOTION	instance_id, hat, value
JOYBUTTONUP	instance_id, button
JOYBUTTONDOWN	instance_id, button
USEREVENT	code

# Lista parziale degli eventi di default.

## I Gestione update e draw

- I `self.screen` è una Surface.
- I Una `Surface` è un'immagine a dimensioni fisse.
- I Una sorta di tela su cui possiamo disegnare ciò che vogliamo.
- I Quando è pronta, la mostriamo all'utente con `display.update()`.


```
YELLOW = (252, 182, 71) # Tupla RGB
```

```
class Game:
```

```
    def update(self):  
        # Per ora tralasciamo l'update.  
        pass
```

```
    def draw(self):  
        # Disegna un cerchio.  
        pg.draw.circle( surface=self.screen,  
                        color= YELLOW,  
                        center=(400, 225),  
                        radius=50)  
  
        pg.display.update() # Aggiorna vista.
```

## I UN GAME LOOP FUNZIONANTE – Il game loop

A questo punto, il programma rimane in vita fino a che non lo chiudiamo cliccando sulla  della finestra.



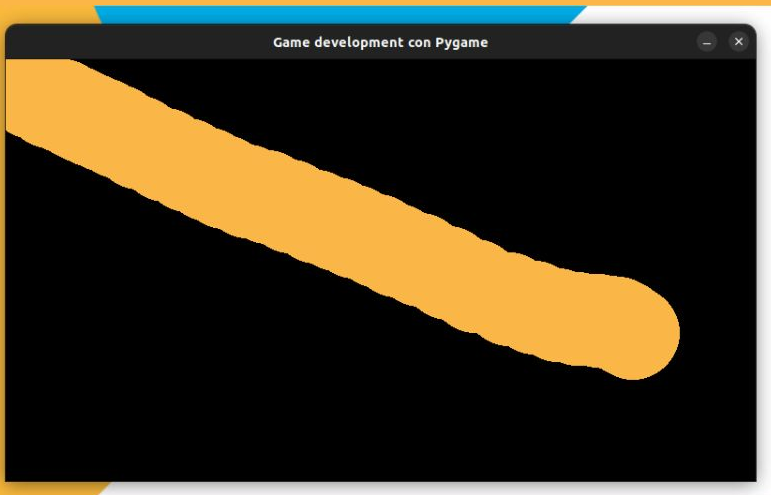
```
class Game:
```

```
    def __init__(self): ...
        self.mouse_pos = (0, 0) # Inizializza l'attributo `mouse_pos`.
        self.run_game_loop()

    def process_events(self):
        for event in pg.event.get(): ...
            elif event.type == pg.MOUSEMOTION: # Catturiamo l'evento `MOUSEMOTION`.
                self.mouse_pos = event.pos      # Posizione del mouse

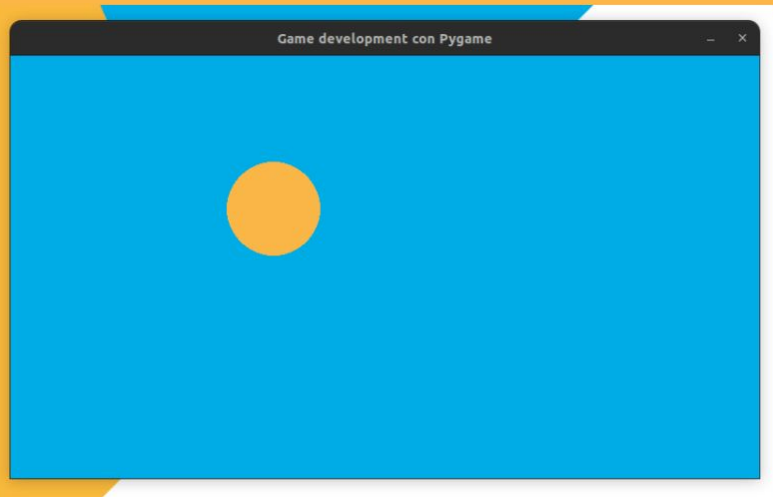
    def draw(self):
        # Disegniamo il cerchio alla posizione del mouse.
        pg.draw.circle(self.screen, YELLOW, self.mouse_pos, 50)
        pg.display.update()
```

## RIPULIRE LA SURFACE – Il game loop

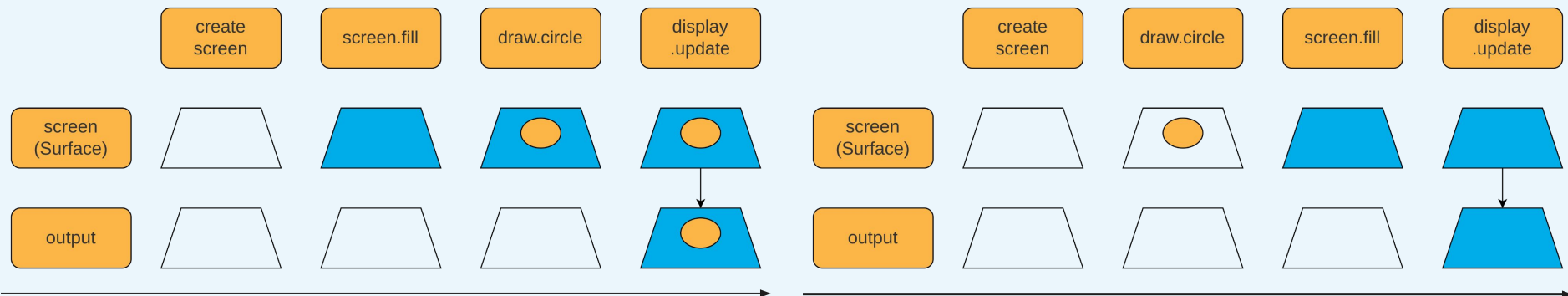


Se non puliamo la  
**Surface**, avremo  
l'effetto "*MS paint*"!

```
# A ogni iterazione, copriamo ciò che  
# è stato disegnato in precedenza.  
+ self.screen.fill(BLUE)  
pg.draw.circle(self.screen, YELLOW, self.mouse_pos, 50)  
pg.display.update()
```



## Ordine di disegno - Il game loop



```
# Prima fill, poi cerchio.
```

```
self.screen.fill(BLUE)
pg.draw.circle(self.screen, YELLOW, (400, 225), 50)

pg.display.update()
```

```
# Prima cerchio, poi fill.
```

```
pg.draw.circle(self.screen, YELLOW, (400, 225), 50)
self.screen.fill(BLUE)

pg.display.update()
```



| Game loop



| Entità?

| ???

| ???

| ???

| ???

| ???

# LA SURFACE E IL RECT

Abbiamo visto come alcune funzioni del modulo `draw` ci permettano di disegnare semplici figure geometriche, ma gli oggetti [Surface](#) e [Rect](#) ci offrono molte più possibilità.

## I Creiamo un quadrato - La Surface e il Rect

```
def init_entities(self): # Da chiamare prima di avviare il game loop.  
  
    # Creiamo una Surface di dimensioni 50x50.  
    self.player = pg.Surface((50, 50))  
  
    # La coloriamo di giallo.  
    self.player.fill(YELLOW)  
  
    # Crea un `Rect` delle stesse dimensioni della `Surface`,  
    # con l'angolo `topleft` posizionato in `(0, 0)`.  
    self.player_rect = self.player.get_rect()
```

## I Disegniamo il quadrato - La Surface e il Rect

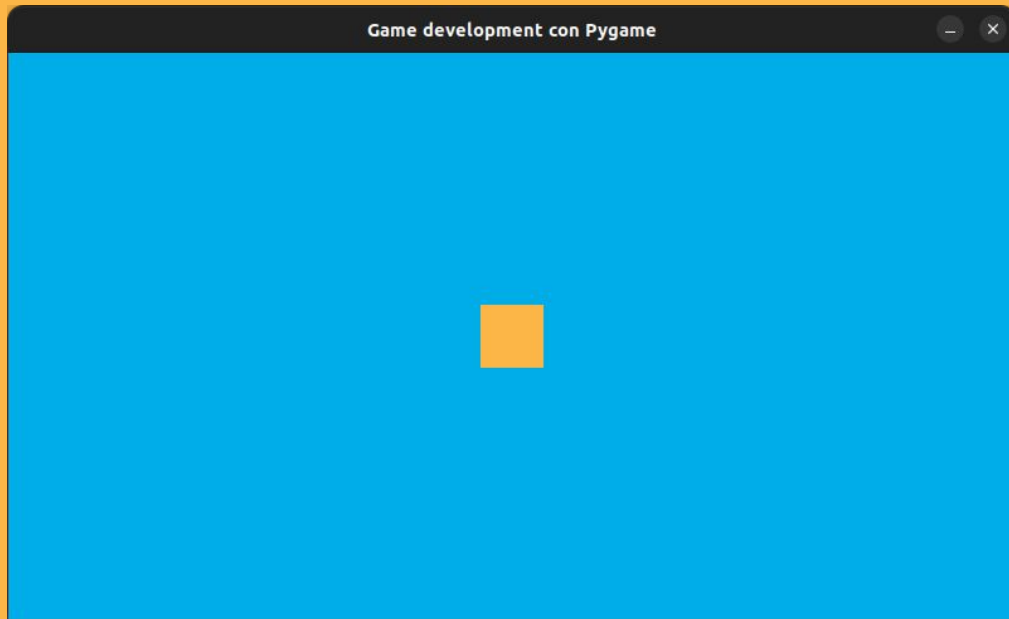
Il metodo `Surface.blit` ci permette di disegnare una **Surface** su un'altra **Surface**, specificando la posizione tramite l'argomento **dest**.

```
def draw(self):  
    self.screen.fill(BLUE)  
    # dest può essere un Rect o una tupla che rappresenta le coordinate.  
    self.screen.blit(source=self.player, dest=self.player_rect)  
    pg.display.update()
```

## I PROPRIO UN BEL QUADRATO – La Surface e il Rect

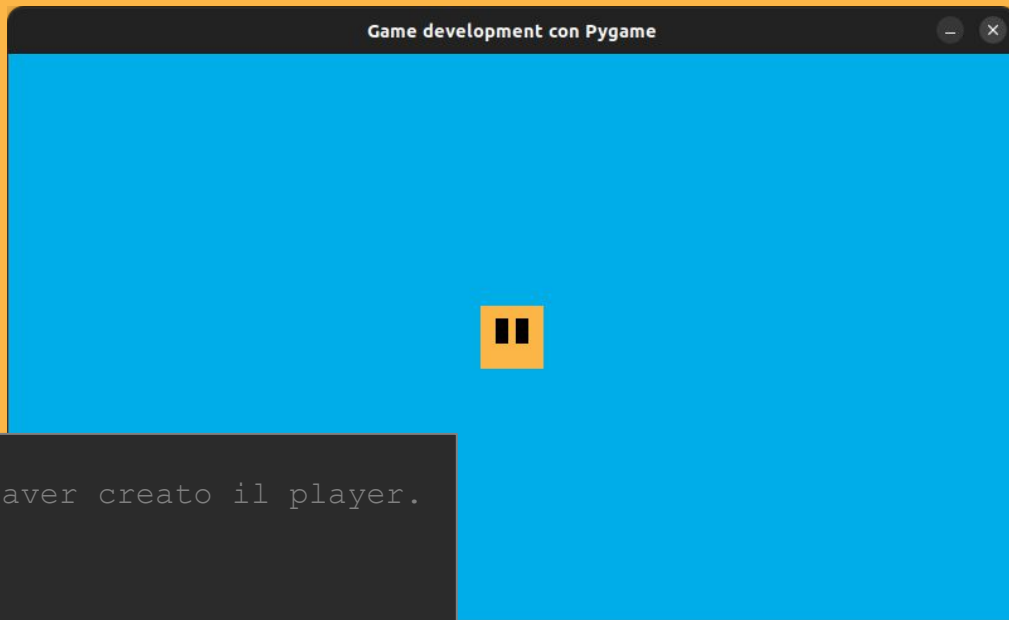
Diciamocelo, è proprio  
un bel quadrato!

Ma cosa ci abbiamo  
guadagnato a usare  
questi oggetti, al posto  
della `draw.rect()`?



## I DISEGNARE SULLE SURFACE – La Surface e il Rect

Su una **Surface** possiamo disegnare qualunque cosa.



```
def init_entities(self): ... # Dopo aver creato il player.  
    eye = pg.Surface((10, 20))  
    eye.fill("black")  
    # Come `Surface.blit`, ma prende un iterabile.  
    self.player.blits([(eye, (12, 10)), (eye, (28, 10))])
```

## I MUOVERE I RECT – La Surface e il Rect

E il **Rect** è un oggetto di cui possiamo modificare facilmente la posizione.



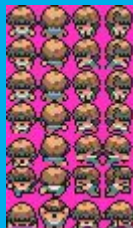
```
def update(self):  
    self.player_rect.x += 1  
    # Se esce dallo schermo, lo ri-posizioniamo a sinistra.  
    if self.player_rect.x > RISOLUZIONE[0]:  
        self.player_rect.right = 0
```

# I DIVERSI TIPI DI IMMAGINE IN COMPUTER GRAPHICS

Sprite



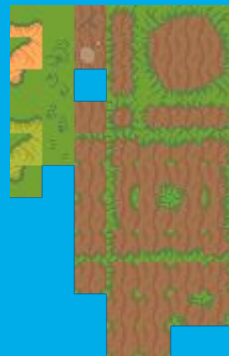
Sprite  
sheet



Tile



Tileset





# LO SPRITE

Lo Sprite è un oggetto che rappresenta le entità di gioco.

Esso ha un attributo `image`, che è una `Surface`, e un attributo `rect`, che è un `Rect`.

Può inoltre essere inserito in dei Group, che ci rendono più semplice lavorare con tanti sprite.



# I Implementazione

```
class Player(pg.sprite.Sprite):

    def __init__(self, *args):
        super().__init__(*args) # *Group.
        self.image = self.get_surface()
        self.rect = self.image.get_rect()

    def get_surface(self) -> pg.Surface: ...
        # Crea la surface del player.

    def update(self): ...
        # Logica di movimento.
```

```
class Game:

    def init_entities(self):
        # Creiamo un gruppo per le entità.
        self.entities = pg.sprite.Group()
        # Creiamo il player.
        self.player = Player(self.entities)

    def update(self):
        # Chiama la update di tutte le entità.
        self.entities.update()

    def draw(self):
        self.screen.fill(BLUE)
        # Disegna tutte le entità.
        self.entities.draw(self.screen)
        pg.display.update()
```

```

class Player(pg.sprite.Sprite):
    speed = 0.4

    def __init__(self, *args): ...
        self.dir = pg.math.Vector2()

    def update(self):
        self.update_dir()
        self.move()

    def update_dir(self):
        # Mapping dei tasti. { key : bool }.
        keys = pg.key.get_pressed()
        # bool convertiti implicitamente a int.
        self.dir.x = keys[pg.K_RIGHT] - keys[pg.K_LEFT]
        self.dir.y = keys[pg.K_DOWN] - keys[pg.K_UP]

    def move(self):
        if self.dir.magnitude() == 0:
            return
        self.rect.move_ip(self.dir * self.speed)

```

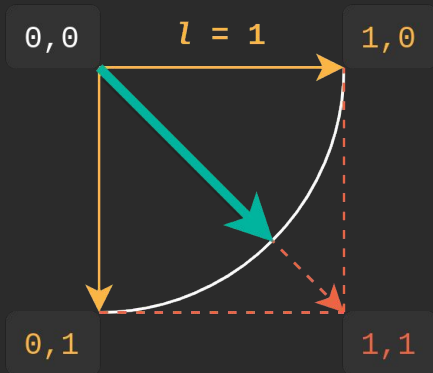
## Utilizziamo l'input utente

### Lo Sprite

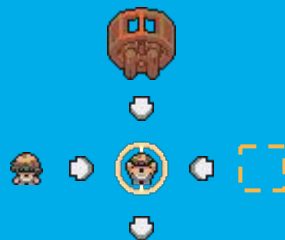


# BUG COMUNE: NORMALIZZAZIONE VETTORE – Lo Sprite

```
def move(self):  
    if self.dir.magnitude() == 0:  
        return  
    # Rende la lunghezza del vettore == `1`.  
    self.dir.normalize_ip()  
    self.rect.move_ip(self.dir * self.speed)
```



| Game loop



| Sprite

| Esperienza  
omogenea?

?



?

| ???



| ???

?



?

| ???



| ???

?

# GESTIONE DEL FRAME RATE



Se provassimo ad eseguire il gioco su pc diversi, vedremo che il **player** si muove a velocità diverse.

Questo perché al momento non abbiamo imposto alcun limite di framerate al *game loop*.

# SOLUZIONI

- Eseguire logica (input + update), e rendering in modo asincrono, per tenere la logica costante.
- Fare i conti in base al tempo trascorso dall'ultimo frame.

# I Gestione del framerate Implementazione

```
class Game:
```

```
    def __init__(self): ...
        self.clock = pg.time.Clock()
        self.run_game_loop()

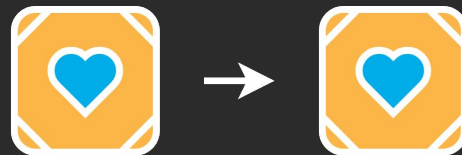
    def run_game_loop(self):
        while True:
            # Limita il framerate e restituisce il
            # tempo trascorso dall'ultima chiamata.
            delta_time = self.clock.tick(60)
            self.process_events(delta_time)
            self.update(delta_time)
            self.draw()

    def update(self, dt: int):
        self.entities.update(dt)
```

```
class Player(pg.sprite.Sprite):
```

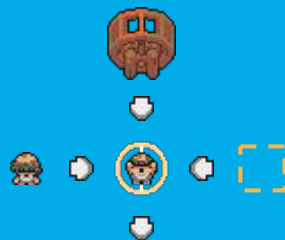
```
    def update(self, dt):
        self.update_dir(dt)
        self.move(dt)

    def move(self, keys, dt): ...
        # Moltiplicando per il delta tempo,
        # lo spostamento rimane costante a
        # prescindere dal framerate.
        self.rect.move_ip(self.dir * self.speed * dt)
```





| Game loop



| Sprite

| Framerate



?

| Livelli?



?

| ???



?

| ???



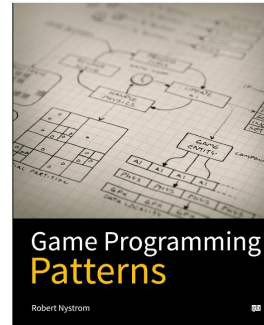
?

| ???

# LO STATO

- Usato per gestire momenti di gioco diversi.
- Utilizziamo una **finite-state machine**.

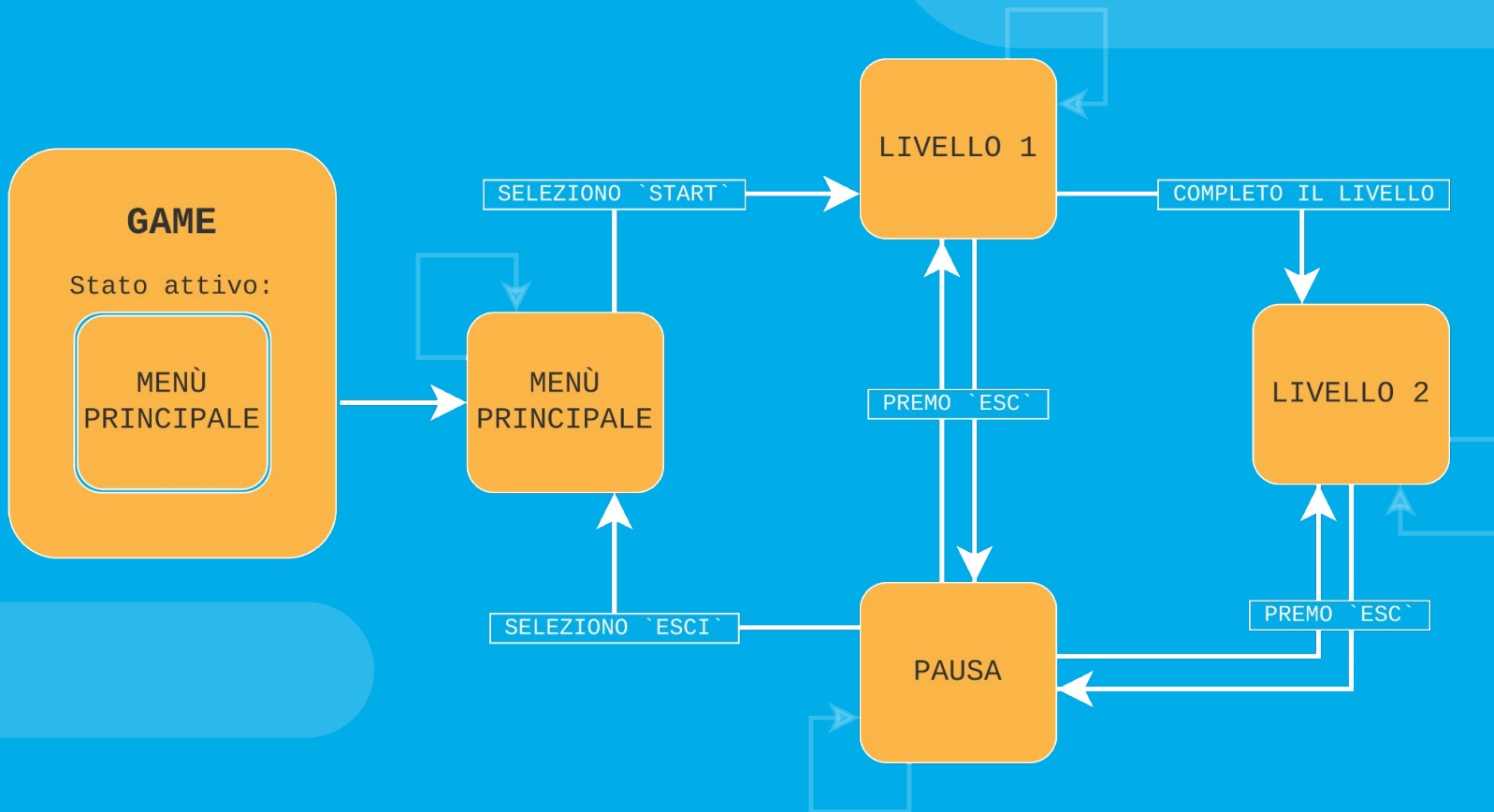
Per approfondire:



## State

[Game Programming Patterns](#) / [Design Patterns Revisited](#)

# FINITE-STATE MACHINE - Lo stato



# I Implementazione

```
class State:
```

```
    def __init__(self, game): ...
    def process_event(self, event, dt): ...
    def update(self, dt): ...
    def draw(self): ...
```

```
class Menu(State): ...
```

```
class World(State): ...
```

```
class Pause(State): ...
```

```
class Game:
```

```
    def __init__(self): ...
        self.states = {"menu": Menu(self),
                        "play": World(self),
                        "pause": Pause(self) }

        self.active_state = self.states["menu"]
        ...

    def process_events(self, dt):
        for event in pg.event.get(): ...
            self.active_state.process_event(event, dt)

    def update(self, dt):
        self.active_state.update(dt)

    def draw(self):
        self.active_state.draw()
        pg.display.update()
```

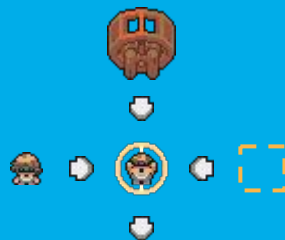
# Implementazione mappa



```
world_map = ["WWWWWWWWWWWWWWWWWW",
             "WWWWWWW  WWWWWWWW",
             "WW          WW   WW",
             "W  W W P    W EW",
             "   E                ",
             "   W W        EW   ",
             "                   ",
             "W  W W          W  W",
             "WW    E  WW   WW",
             "WWWWWWW  WWWWWWWW",
             "WWWWWWW  WWWWWWWW"]
```

```
for row_index, row in enumerate(world_map):
    for col_index, col in enumerate(row):
        x = col_index * TILESIZE
        y = row_index * TILESIZE
        # Crea/posiziona entità.
        # Disegna lo sfondo su una `Surface`
        # da riutilizzare.
```

| Game loop



| Sprite

| Framerate



| Stato

| Oggetti solidi?

?



?

| ???



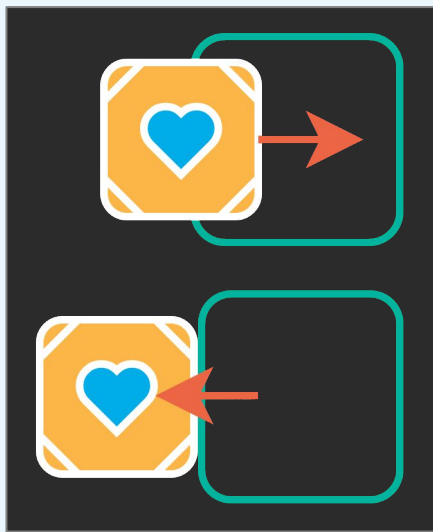
?

| ???

## I Collisioni

Un'alternativa a

`Rect.colliderect()` è usare i  
metodi del modulo `sprite` per le  
collisioni tra **Group**.

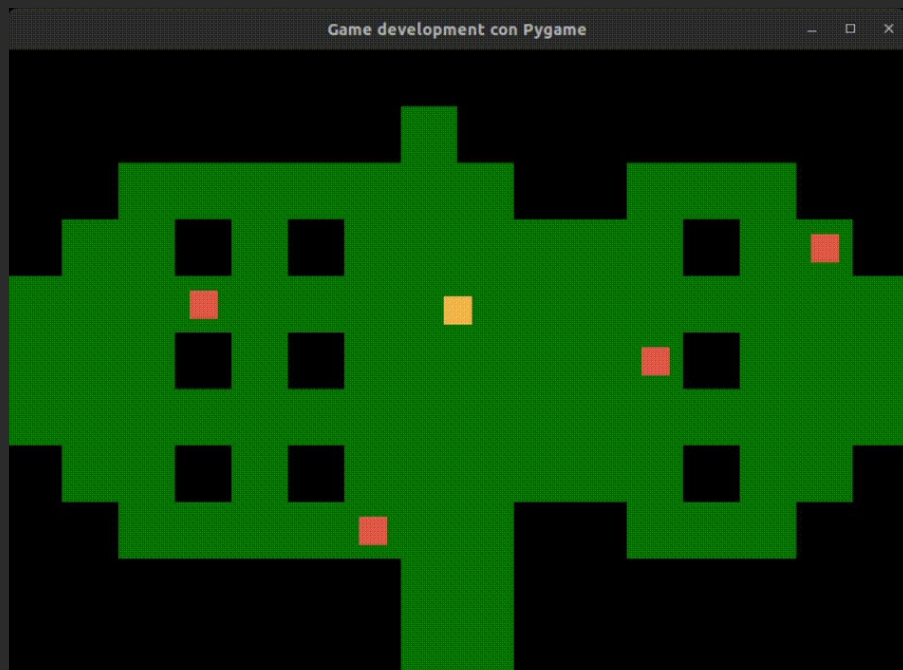


```
def update(self, dt):
    self.update_dir(dt)          # Aggiorna direzione.
    self.move(dt)                # Muove il Player.
    self.collide_entities()      # Controlla collisioni.

def collide_entities(self):
    if self.dir.magnitude() == 0:
        return
    x, y = self.dir.xy

    for entity in self.colliding_entities:
        if self.rect.colliderect(entity.rect):
            if x > 0:
                self.rect.right = entity.rect.left
            elif x < 0:
                self.rect.left = entity.rect.right
            if y > 0:
                self.rect.bottom = entity.rect.top
            elif y < 0:
                self.rect.top = entity.rect.bottom
```

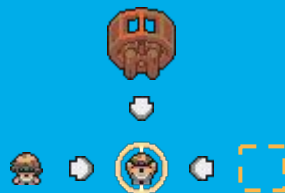
## BUG COMUNE: TELETRASPORTO ALL'ANGOLO – Collisioni



Vedi demo di implementazione corretta.



| Game loop



| Sprite

| Framerate



| Stato



| Collisioni



| Azione?



| ???



## I Interazioni Al per i nemici

```
class World(State): ...

    def update(self, dt):
        # `self.actors` è il `Group` che
        # contiene tutti gli `Actor`.
        self.actors.update(dt)

        attacking_enemies = pg.sprite.spritecollide(
            self.player, self.enemies, False)

        for enemy in attacking_enemies:
            enemy.hit(self.player)
```

# La classe `Actor` deriva da `Sprite`, ed  
# è la classe base di `Player` e `Enemy`.

```
class Enemy(Actor): ...

    speed = .3
    view_range = 250
    max_hp = 1
    damage = 1

    def __init__(self, player, *args): ...
        self.player = player

    def update_dir(self, dt): # Chiamata nella `update`.
        px, py = self._player.center
        ex, ey = self.center
        self.dir.xy = px - ex, py - ey

        if self.dir.magnitude() > self.view_range:
            self.dir.xy = 0, 0

    def hit(self, entity: Actor):
        entity.suffer_damage(self.damage)
```

# I Interazioni

## Attacco

```
class World(State): ...

    def process_event(self, event: pg.event.Event, dt: int):
        if event.type == pg.KEYDOWN: ...
            if event.key == pg.K_SPACE:
                self.attacks.add(self.player.attack())

    def update(self, dt): ...
        if self.player.is_attacking():
            self.attacks.update()
            attacked_enemies = pg.sprite.groupcollide(
                self.attacks, self.enemies, False, False)

            for attack, enemies in attacked_enemies.items():
                for enemy in enemies:
                    attack.hit(enemy)
```

```
class Attack(pg.sprite.Sprite): ...
    damage: int = 1
    animation_time: int = 200

    def __init__(self, player: Player, *groups): ...
        self.animation_start = pg.time.get_ticks()

    def update(self):
        self.move() # Sposta l'attacco vicino al player.
        timeout = self.animation_start + self.animation_time
        if pg.time.get_ticks() > timeout:
            self.player.attack = None
            self.kill()

class Player(Actor): ...

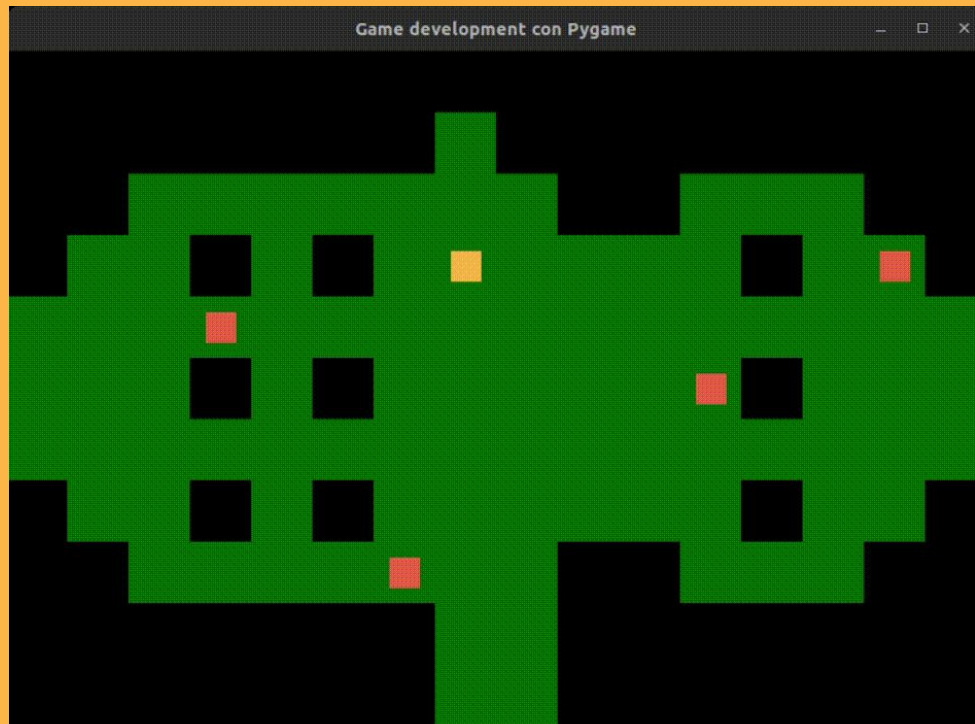
    def attack(self) -> Attack:
        if not self.is_attacking():
            self.attack = Attack(self)
        return self.attack

    def is_attacking(self):
        return bool(self.attack)
```

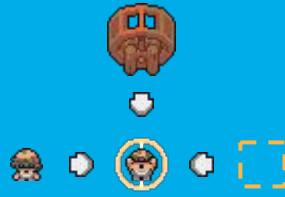
## ■ ABBIAMO UN PROTOTIPO! – Interazioni

Abbiamo una mappa,  
delle entità, degli stati, e  
ora anche l'azione!

Le *core mechanics* ci  
son tutte!



| Game loop



| Sprite

| Framerate



| Stato

| Collisioni

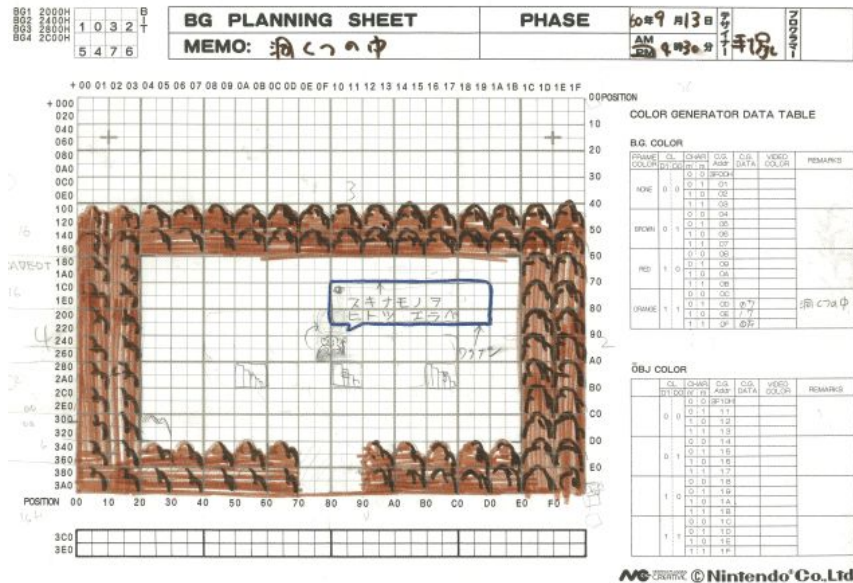


| Interazioni

| Grafica e  
musica?



# ASSETS



- Caricare immagini.
- Riprodurre animazioni.
- Riprodurre suoni.
- Scrivere testi.

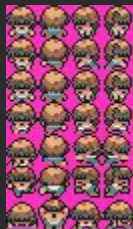
# Immagini e animazioni

```
class Spritesheet:
```

```
def __init__(self, filename: str):
    self.sheet = pg.image.load(filename).convert_alpha()
    size_x, size_y = self.sheet.get_size()
    self.rows = size_y // TILESIZE
    self.cols = size_x // TILESIZE

def img_at(self, col: int, row: int) -> pg.Surface:
    # Restituisce il tile alle coordinate specificate.

def images_at_col(self, col: int) -> list[pg.Surface]:
    return [self.img_at(col, row) for row in range(self.rows)]
```



```
# Classe derivata da Sprite.
def __init__(self, ..., *args): ...
    self.movement_animations = self._init_animations()
    self.set_animation("down")

def _init_animations(self):
    spritesheet = Tileset("assets/images/spritesheet.png",
TILESIZE)
    return {"down": spritesheet.images_at_col(0),
            "up": spritesheet.images_at_col(1),
            "left": spritesheet.images_at_col(2),
            "right": spritesheet.images_at_col(3)}

def set_animation(self, animation: str):
    self.animation: list[pg.Surface] =
        self.movement_animations[animation]
    self.current_animation_idx = 0

def update(self, dt): ...
    # Riproduce l'animazione, aggiornando
    self.current_animation_idx.

@property
def image(self) -> pg.Surface:
    return self.animation[self.current_animation_idx]
```

## Assets Suoni

```
# Carica la musica di sottofondo.
pg.mixer.music.load(filename="music/theme.ogg")

# Setta il volume della musica di sottofondo.
pg.mixer.music.set_volume(4)

# `-1` == Loop infinito.
pg.mixer.music.play(loops=-1)

# Termina la riproduzione.
pg.mixer.music.stop()

# Inizializza un suono.
attack_sound = pg.mixer.Sound("sounds/Attack.wav")

# Riproduce il suono.
attack_sound.play()
```

## Assets Testi

```
# Inizializza font.
font = pg.font.Font(filename="fonts/MyFont.ttf",
                    size=42)

comic_sans = pg.font.SysFont("comicsansms", 60)
default_font = pg.font.Font(None, 20)

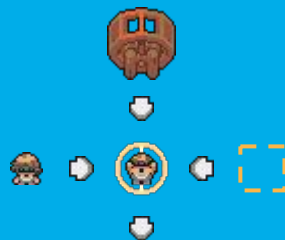
# Crea `Surface` con scritta.
font_surf = font.render(text="PAUSA", antialias=False,
                        color=YELLOW, background=None)

font_rect = font_surf.get_rect()

# Disegno la surface su `self.screen`.
self.screen.blit(font_surf, font_rect)
```



| Game loop



| Sprite

| Framerate



| Stato

| Collisioni



| Interazioni

| Assets



# LA NOSTRA DEMO È PRONTA!



# Luca Amirante

lamirante@develer.com



[www.develer.com](http://www.develer.com)