**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

# BÁO CÁO ĐỒ ÁN

## Môn học: ngôn ngữ lập trình Python

## Đề tài: Xây dựng ứng dụng game chiến đấu 2D

**Giảng viên: TS Trịnh Tấn Đạt**

**Thành viên nhóm:**

| STT | Họ và tên | Mã số sinh viên |
|---|---|---|
| 1 | Trương Nguyễn Minh Hiếu | 3122410118 |
| 2 | Lê Chí Hào | 3122410096 |
| 3 | Bùi Quang Minh Hiếu | 3122410112 |
| 4 | Hoàng Đình Hoàn | 3122410123 |

Thành phố Hồ Chí Minh, ngày , tháng , năm 2024

# Nhận xét, đánh giá của giảng viên

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

# LỜI CẢM ƠN

Để hoàn thành được bài tiểu luận này, em xin chân thành cảm ơn và gửi lời tri ân sâu sắc đến quý thầy, cô thuộc bộ môn ngôn ngữ lập trình python thuộc khoa Công nghệ thông tin, trường Đại học Sài Gòn đã tạo cơ hội cho em được học tập, nghiên cứu và tích lũy kiến thức để thực hiện bài báo cáo đồ án. Trên hết, em xin được gửi lời cảm ơn chân thành nhất đến giảng viên, TS Trịnh Tấn Đạt đã tận tình chỉ dẫn và đưa ra nhiều lời khuyên bổ ích giúp em hoàn thành bài báo cáo này một cách tốt nhất.

Do kiến thức của bản thân còn nhiều hạn chế và thiếu kinh nghiệm thực tiễn nên nội dung bài báo cáo khó tránh khỏi những thiếu sót. Chúng em rất mong nhận được những lời góp ý thêm từ quý thầy cô, để chúng em học hỏi thêm được nhiều kỹ năng, kinh nghiệm trong quá trình học tập và sẽ hoàn thành tốt hơn cho những bài báo cáo sắp tới.

Chúng em xin chân thành cảm ơn thầy.

Trân trọng.

# MỤC LỤC

# PHẦN I: CƠ SỞ LÝ THUYẾT

## 1. Giới thiệu về ngôn ngữ python

### 1.1. Ngôn ngữ Python là gì?

Python là một ngôn ngữ lập trình bậc cao, được tạo ra bởi Guido van Rossum vào những năm 1980 và phát triển tiếp theo trong thập niên 1990. Python là một ngôn ngữ lập trình đơn giản, dễ học và dễ sử dụng, có thể được sử dụng để xây dựng các ứng dụng máy tính, trang web, các hệ thống đám mây, trí tuệ nhân tạo, máy học và các ứng dụng phân tích dữ liệu. Python cũng là một ngôn ngữ lập trình đa nền tảng, có thể chạy trên các hệ điều hành khác nhau như Windows, Linux và MacOS. Python có một cộng đồng lớn và đa dạng, với hàng ngàn thư viện và framework hỗ trợ cho việc phát triển ứng dụng.

### 1.2. Lợi ích của Python

- **Dễ học và sử dụng:** Python có cú pháp đơn giản và dễ hiểu, giúp người mới bắt đầu học lập trình dễ dàng tiếp cận và làm quen với các khái niệm lập trình cơ bản.

  Đa năng: Python có thể được sử dụng để phát triển các loại ứng dụng khác nhau như ứng dụng máy tính, web, trí tuệ nhân tạo, máy học, phân tích dữ liệu, đám mây, hệ thống nhúng và nhiều hơn nữa.
- **Cộng đồng lớn và đa dạng:** Python có một cộng đồng phát triển lớn và đa dạng, với hàng ngàn thư viện và framework hỗ trợ cho việc phát triển ứng dụng.
- **Tính tương thích cao:** Python là một ngôn ngữ đa nền tảng, có thể chạy trên nhiều hệ điều hành khác nhau như Windows, Linux và MacOS.
- **Hiệu suất cao:** Python có thể được tối ưu hóa để đạt được hiệu suất cao, đặc biệt là khi sử dụng các thư viện và framework được viết bằng C/C++.
- **Dễ dàng mở rộng:** Python có tính mở rộng cao, cho phép người dùng tạo ra các module và thư viện riêng để mở rộng chức năng của Python.
- **Tương tác tốt với các ngôn ngữ khác:** Python có thể tương tác tốt với các ngôn ngữ khác như C/C++, Java, và .NET, giúp cho việc tích hợp các phần mềm và dịch vụ khác nhau trở nên dễ dàng hơn.

### 1.3. Ứng dụng của Python

❖ Dưới đây là một số ví dụ về các ứng dụng của Python:

- **Phát triển ứng dụng web:** Python có thể được sử dụng để phát triển các ứng dụng web, với các framework như Django, Flask và Pyramid.
- **Trí tuệ nhân tạo và máy học:** Python là một trong những ngôn ngữ phổ biến nhất trong lĩnh vực trí tuệ nhân tạo và máy học, với các thư viện như TensorFlow, Keras và PyTorch.
- **Phân tích dữ liệu**: Python có các thư viện như Pandas và NumPy để xử lý và phân tích dữ liệu, và Matplotlib và Seaborn để tạo đồ thị và biểu đồ.
- **Game development:** Python có thể được sử dụng để phát triển game với các thư viện như Pygame và PyOpenGL.
- **Đám mây:** Python có thể được sử dụng để quản lý và triển khai các hệ thống đám mây với các framework như OpenStack và Ansible.
- **Hệ thống nhúng:** Python có thể được sử dụng để phát triển các hệ thống nhúng, với các framework như MicroPython và CircuitPython.
- **Ứng dụng máy tính:** Python có thể được sử dụng để phát triển các ứng dụng máy tính trên nhiều nền tảng khác nhau, với các thư viện như PyQt và wxPython.

# PHẦN II: MỞ ĐẦU

## 1. Giới thiệu đề tài

Trong khuôn khổ đồ án này, chúng em đã thực hiện việc xây dựng một trò chơi 2D chiến đấu đơn giản nhằm mục đích nghiên cứu và áp dụng các kỹ thuật lập trình vào việc phát triển game. Trò chơi được xây dựng sử dụng ngôn ngữ lập trình Python và chủ yếu được lập trình dựa trên thư viện Pygame, nhằm mục tiêu tạo ra một trải nghiệm giải trí thú vị , mới mẻ cho người chơi.

## 2. Mục đích và Mục tiêu

❖ **Mục đích:**

- Nắm chắc được được kỹ năng và kiến thức về lập trình.
- Tìm hiểu về thư viện Pygame trong ngôn ngữ lập trình Python.
- Cũng cố, áp dụng, nâng cao kiến thức đã được học.
- Nắm bắt được quy trình làm game cơ bản.

❖ **Mục Tiêu:**

- Xây dựng một trò chơi 2D đơn giản với giao diện đồ họa hấp dẫn.
- Thử nghiệm và áp dụng các kỹ thuật lập trình Python vào việc phát triển game.
- Tạo ra một trò chơi dễ dàng tiếp cận và chơi thú vị cho người dùng.
- Nâng cao kỹ năng lập trình và hiểu biết về phát triển game của nhóm thực hiện.

## 3. Phương Pháp Thực Hiện

a. **Thiết kế Game:** Trước hết, chúng em đã đặt ra ý tưởng và thiết kế cơ bản một dòng game chiến đấu cơ bản. Đầu tiên là lối chơi - thể loại chiến đấu qua màn . Tiếp theo là hình ảnh nhân vật game , quái và bản đồ.
b. **Lập trình:** Sử dụng ngôn ngữ lập trình Python cùng thư viện Pygame, chúng em đã tiến hành lập trình các tính năng của trò chơi, bao gồm xử lý sự kiện, vẽ đồ họa, và quản lý trạng thái game.
c. **Kiểm thử và sửa lỗi**: Sau khi hoàn thành, chúng em đã tiến hành kiểm thử trò chơi để phát hiện và sửa lỗi, đảm bảo trải nghiệm chơi game là mượt mà và không gặp phải vấn đề kỹ thuật.

## 4. Giới thiệu thư viện được sử dụng trong đề tài

### 4.1. Những thư viện được sử dụng

**Tkinter**: Tkinter là một thư viện mặc định của Python cho phép tạo giao diện đồ họa người dùng (GUI). Tkinter được xây dựng dựa trên toolkit Tk, một toolkit đa nền tảng để xây dựng giao diện người dùng đồ họa. Với Tkinter, tạo các phần tử như nút bấm, khung, cửa sổ, đối tượng văn bản, hộp thoại. Nó cung cấp một cách tiếp cận tương đối dễ dàng cho việc tạo các ứng dụng desktop đơn giản, đặc biệt là các ứng dụng có giao diện người dùng đơn giản.

**Customtkinter:** là một thư viện Python cung cấp các tùy chọn tùy chỉnh và chức năng bổ sung cho bộ công cụ Tkinter GUI. Nó được xây dựng dựa trên Tkinter và cho phép các nhà phát triển tạo ra các giao diện người dùng tương tác.

**XlsxWriter:** là một thư viện Python cho phép tạo các tệp Excel (.xlsx) với các định dạng, kiểu dữ liệu và tính năng khác nhau. Nó cho phép bạn tạo các bảng tính Excel từ dữ liệu Python, định dạng các ô, định dạng cột và hàng, thêm các biểu đồ và hình ảnh, và nhiều hơn nữa.

**Openpyxl** là một thư viện Python dùng để đọc và ghi các tệp Excel. Thư viện này cho phép người dùng truy cập và điều chỉnh các giá trị trong các ô, thêm và xóa các bảng tính, định dạng văn bản và thêm các biểu đồ, hình ảnh và các đối tượng khác vào tệpExcel.

**Pandas** là một thư viện mã nguồn mở cho ngôn ngữ lập trình Python được sử dụng để phân tích và xử lý dữ liệu. Pandas cho phép người dùng đọc, ghi và sửa đổi dữ liệu từ các nguồn khác nhau như tệp CSV, Excel, SQL, HTML và nhiều nguồn dữ liệu khác.

**NumPy** (Numerical Python) là một thư viện tính toán khoa học cho Python. NumPy cung cấp các đối tượng và hàm để làm việc với mảng và ma trận đa chiều. NumPy là một thư viện quan trọng và được sử dụng rộng rãi trong các lĩnh vực như khoa học dữ liệu, xử lý ảnh, máy học và nhiều lĩnh vực khác.

**Tkcalendar** là một module của Python cung cấp một widget lịch cho toolkit GUI Tkinter. Nó cho phép dễ dàng thêm một lịch vào ứng dụng Tkinter của bạn và tùy chỉnh giao diện và hành vi của nó. Với Tkcalendar, dùng để thể hiển thị các ngày, tuần, tháng hoặc năm với các chức năng như chọn ngày. Nó cũng cung cấp cho người dùng một nơi để chọn ngày từ lịch, và các sự kiện có thể được kích hoạt khi người dùng chọn ngày nhất định.

**Pillow** là một thư viện Python được sử dụng để xử lý ảnh. Thư viện này cung cấp các công cụ để mở, chỉnh sửa và lưu các tệp ảnh, bao gồm các định dạng phổ biến như JPEG, PNG, BMP và GIF. Nó được xây dựng dựa trên thư viện PIL (Python Imaging Library), nhưng được phát triển tiếp từ năm 2011.

# PHẦN III: NỘI DUNG

## I. Đôi nét về game chiến đấu 2D

### 1. Giới Thiệu Về Game Chiến Đấu 2D

Trong thế giới của game điện tử, thể loại game chiến đấu luôn là một trong những thể loại được ưa chuộng nhất. Game chiến đấu 2D là một dạng game nổi tiếng và có lịch sử dài đối với cộng đồng game thủ. Dưới đây là một sự giới thiệu về thể loại game này:

**a. Bản chất của game chiến đấu 2D**

- **Lối chơi**: Trong game chiến đấu 2D, người chơi thường điều khiển một nhân vật và tham gia vào các trận đấu đối kháng với những đối thủ khác. Lối chơi thường xoay quanh việc sử dụng các kỹ năng và chiêu thức để đánh bại đối thủ.
- **Đồ họa**: Đa phần game chiến đấu 2D được thiết kế với đồ họa 2 chiều, thường là từ góc nhìn ngang hoặc đứng. Điều này tạo ra một phong cách đồ họa đơn giản nhưng vẫn đảm bảo sự thú vị và mượt mà trong trải nghiệm chơi game.
- Nhân vật và vũ khí: Game chiến đấu 2D thường có một loạt các nhân vật với kỹ năng và vũ khí đặc biệt riêng. Người chơi có thể lựa chọn nhân vật yêu thích của mình và sử dụng các vũ khí độc đáo để chiến đấu.

**b. Đặc điểm nổi bật**

- **Hành động nhanh nhẹn**: Game chiến đấu 2D thường được thiết kế với tốc độ nhanh, yêu cầu người chơi phản xạ nhanh và phản ứng linh hoạt.

- **Cấu trúc đơn giản:** Dù có thể có sự đa dạng về nhân vật và vũ khí, nhưng game chiến đấu 2D thường có cấu trúc đơn giản và dễ tiếp cận.
- **Yếu tố chiến lược:** Mặc dù nhiều game chiến đấu 2D chủ yếu là về hành động, nhưng cũng có các yếu tố chiến lược như quản lý kỹ năng và tư duy chiến thuật.

**c. Vai trò trong ngành công nghiệp game**

Game chiến đấu 2D không chỉ là một thể loại giải trí mà còn là một phần quan trọng của ngành công nghiệp game. Với sự đa dạng về nhân vật, vũ khí và cốt truyện, game chiến đấu 2D luôn thu hút sự chú ý của cả những game thủ mới và những người chơi kỳ cựu. Đồng thời, thể loại này cũng là một nền tảng tuyệt vời cho các giải đấu eSports và cộng đồng game thủ chuyên nghiệp.

### 2. Mục tiêu của game chiến đấu 2D

Mục tiêu của game chiến đấu 2D là điều khiển nhân vật vượt qua các chướng ngại vật để qua màn. Sẽ có các con quái đến chiến đấu với bạn, nếu bạn đánh thắng sẽ đi đến cửa qua màn, nếu bạn chiến đấu thua quái, trò chơi sẽ kết thúc.

## II. Yêu cầu với đồ án

Yêu cầu chương trình của chúng ta ở đây là làm sao thiết kế được một chương trình game giống (hoặc gần giống) như game chiến đấu 2D trên dựa vào các phương pháp lập trình đã học. Cụ thể là sử dụng phương pháp lập trình Python đang học kết hợp với các mẫu thiết kế và các giải thuật toán học.

# III. Thực nghiệm và phân tích kết quả

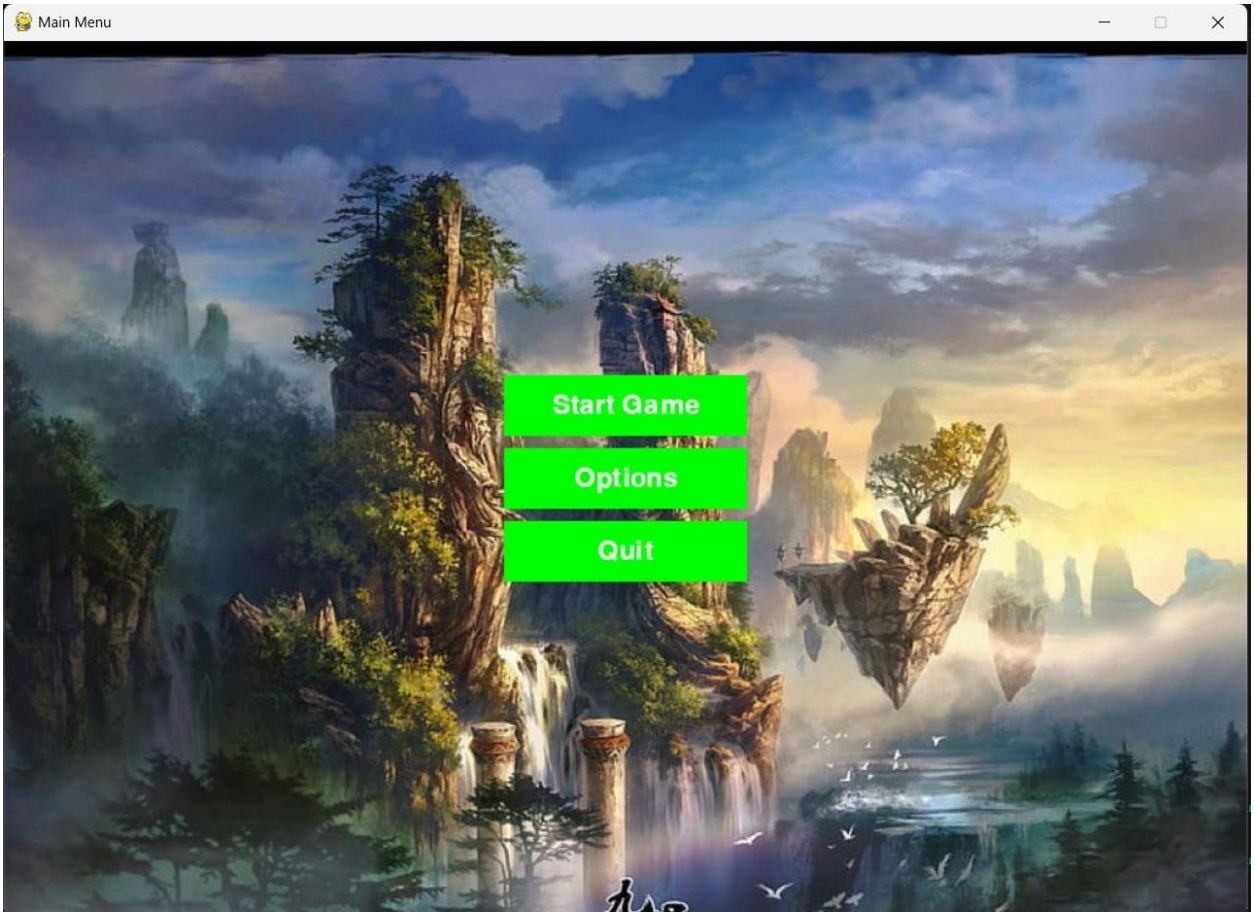
## a. Xây dựng hình ảnh

### ❖ Giao diện

➢ Bố cục chơi game gồm 1 màn hình menu và 1 screen chính.

**- Một màn hình menu:**

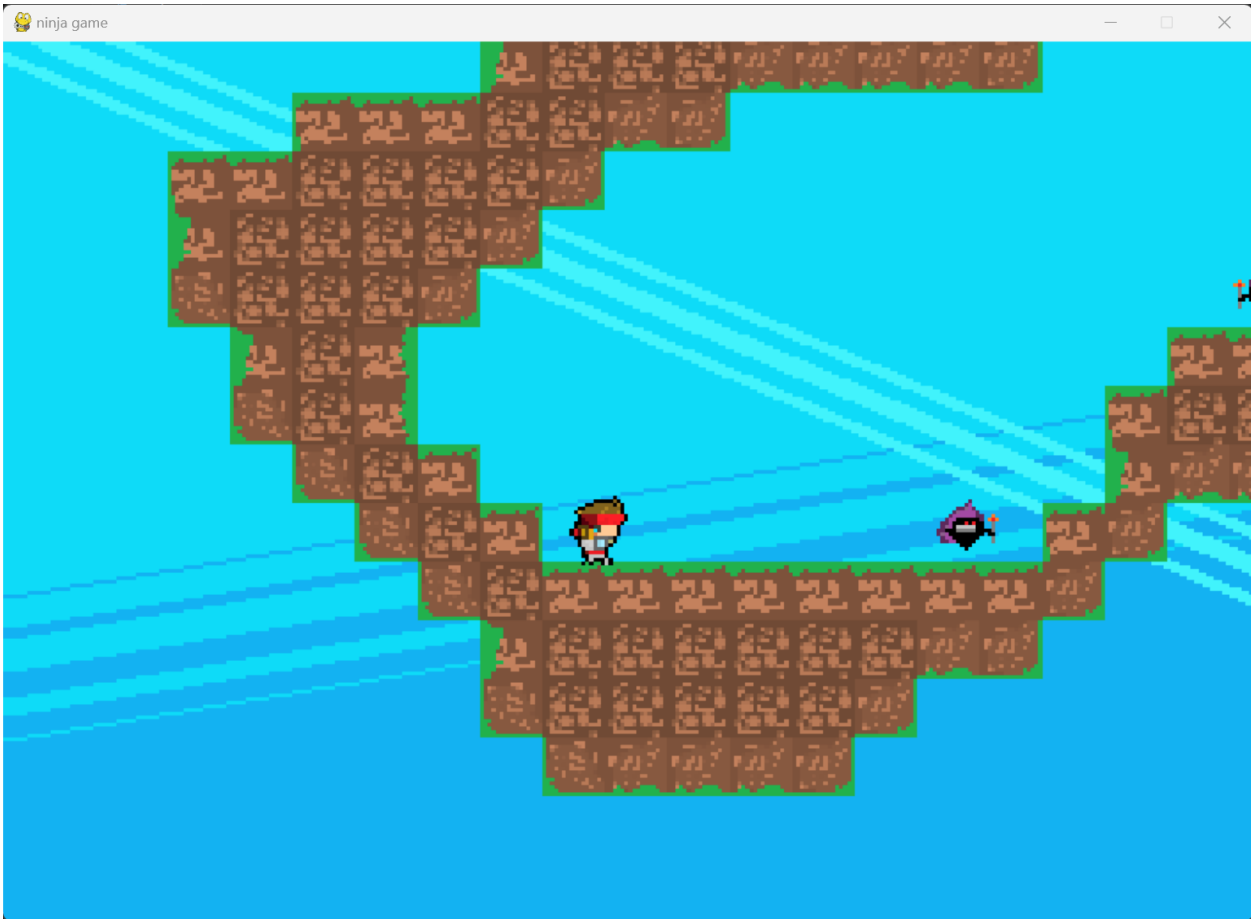
▪ Kích thước: SCREENWIDTH = 1024(px), SCREENHIGH= 720(px),




Ở trên màn hình menu sẽ có 3 button hiển thị các chức năng:

- Button Start Game
- Button Options
- Button Quit

- **Một screen chính:**
  - Kích thước: width = 1024(px), height = 720(px),

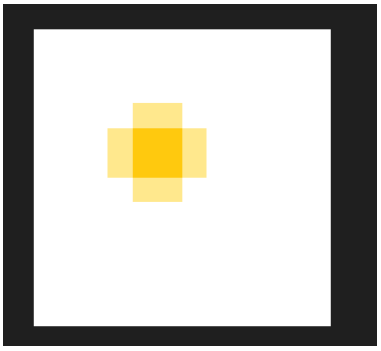



- Hình ảnh nhân vật

Nhân vật là một ninja được vẽ bằng pixal có kích thước là chiều dài: 19 ô và rộng là 15.

Ninja với chiêu thức lướt qua kẻ địch để tiêu diệt kẻ địch một cách nhanh chóng và gọn gàng. Khi lướt qua kẻ địch, ninja sẽ phóng ra các hạt màu vàng.

- Hình ảnh các hạt



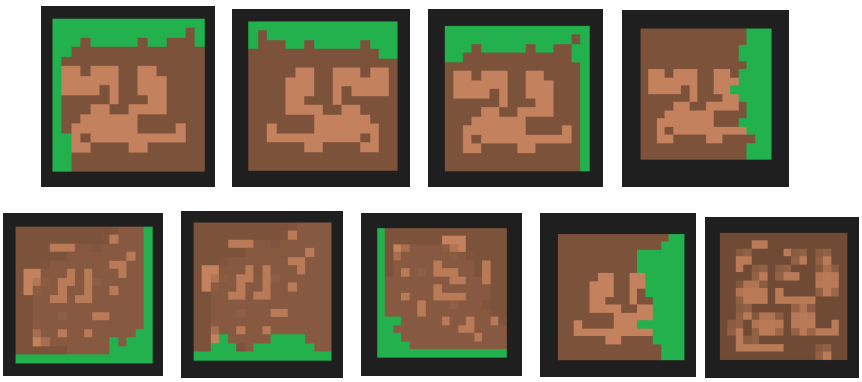
- Hình ảnh quái thú

Quái thú được vẽ với kích thước là chiều dài 16 ô với chiều rộng 16 ô với hình ảnh một con quái thú màu tím tay cầm vũ khí và bắn ra các hạt đạn.

- Hình ảnh của các hạt đạn của quái thú. Các hạt đạn được vẽ với kích thước là chiều dài là 4 ô và chiều rộng là 6 ô.

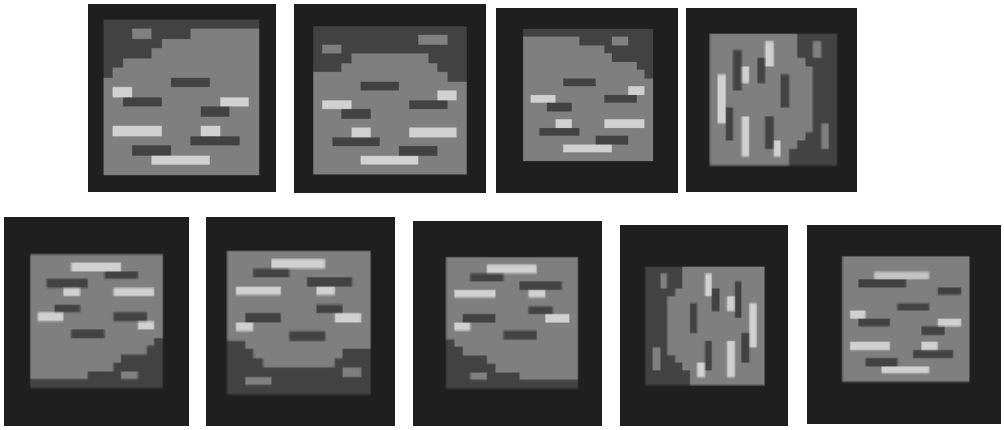
- Hình ảnh map
  - Đường đất

Có tất cả 9 hình đường đất hiển thị trong game. Các hình đường đất có kích thước là chiều dài 16 ô và chiều rộng 16 ô.



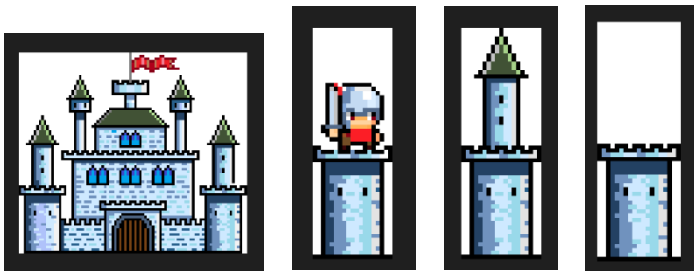
- Đường đá

Có tất cả 9 hình đường đá hiển thị trong game. Các hình đường đá có kích thước là chiều dài 16 ô và chiều rộng 16 ô.



- Các hình ảnh khác
  - Cây

  - Lâu đài



## b. Luật chơi cơ bản

- Dùng các phím ↑, phím ↓, phím ←, phím → để điều khiển nhân vật nhảy lên, đi xuống, sang trái, sang phải.
- Dùng phím X để sử dụng kĩ năng theo từng nhân vật
- Nhân vật phải tránh được các đòn tấn công của quái vật và dùng kĩ năng để tiêu diệt các quái trên bản đồ để đi đến đích
- Sau khi tiêu diệt hết các quái thì sẽ tự động qua màn.

## c. Phân tích

### 1. Cài đặt pygame và các biến

- Mở command prompt trên Windows hoặc terminal trên MacOS rồi nhập vào dòng lệnh:

```
pip install pygame
```

### 2. Hàm main

Dùng để khởi tạo menu bao gồm:

+ Play: khi click vào dung để bắt đầu trò chơi ( mở game.py)

+ Option: Dùng để chọn nhân vật để bắt đầu trò chơi

+ Thoát: Dùng để thoát chương trình

```
pygame.init()
```

### a. Button và clicked trên menu:

```
class Button:
    def __init__(self, x, y, width, height, color, text=''):
        self.rect = pygame.Rect(x, y, width, height)
        self.color = color
        self.text = text
        self.font = pygame.font.Font(None, 32)
```

Đây là phương thức khởi tạo cho một lớp đối tượng trong Python, được sử dụng để tạo ra một phần tử đồ họa cơ bản bằng thư viện Pygame. Hãy phân tích từng tham số:

- **x, y**: Tọa độ của góc trên bên trái của hình chữ nhật.
- **width**, **height**: Kích thước của hình chữ nhật.
- **color**: Màu sắc của phần tử đồ họa.
- **text**: Tuỳ chọn, là văn bản cần hiển thị bên trong phần tử.
- **font**: Phông chữ được sử dụng cho văn bản, được khởi tạo với kích thước mặc định là 32.

```
def draw(self, screen):
    pygame.draw.rect(screen, self.color, self.rect)
    if self.text:
        text_surface = self.font.render(self.text, True, (255, 255, 255))
        text_rect = text_surface.get_rect(center=self.rect.center)
        screen.blit(text_surface, text_rect)

def is_clicked(self, pos):
    return self.rect.collidepoint(pos)
```

- **pygame.draw.rect(screen, self.color, self.rect)**: Vẽ một hình chữ nhật với màu **self.color** lên màn hình **screen**, sử dụng hình chữ nhật đã được xác định trong **self.rect**.
- **if self.text::** Kiểm tra xem có văn bản được chỉ định không.
- **text_surface = self.font.render(self.text, True, (255, 255, 255)):** Tạo một bề mặt văn bản từ **self.text**, sử dụng phông chữ đã được xác định trong **self.font**, với màu trắng (255, 255, 255).
- **text_rect = text_surface.get_rect(center=self.rect.center):** Lấy hình chữ nhật chứa văn bản và căn giữa nó theo tọa độ của hình chữ nhật chứa nó **(self.rect).**
- **screen.blit(text_surface, text_rect):** Vẽ văn bản lên màn hình **screen**, tại vị trí được xác định bởi **text_rect**.

Phương thức này hoạt động bằng cách vẽ một hình chữ nhật và sau đó vẽ văn bản lên trung tâm của hình chữ nhật đó (nếu có).

```
def is_clicked(self, pos):
    return self.rect.collidepoint(pos)
```

Phương thức **is_clicked** này kiểm tra xem một điểm nhất định (**pos**) có nằm trong phần tử đồ họa hay không. Dưới đây là cách nó hoạt động:

- **self.rect.collidepoint(pos):** Phương thức này kiểm tra xem một điểm (**pos**) có nằm trong hình chữ nhật được xác định bởi **self.rect** hay không. Nếu điểm đó nằm trong hình chữ nhật, phương thức sẽ trả về **True**, ngược lại trả về **False**.

Do đó, phương thức **is_clicked** sẽ trả về **True** nếu điểm **pos** nằm trong hình chữ nhật của phần tử đồ họa, và **False** nếu không.

**b. Xử lý cửa sổ và giao diện của menu**

```
SCREEN_WIDTH = 1024
SCREEN_HEIGHT = 720
screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
pygame.display.set_caption("Main Menu")
```

Đoạn mã trên đặt kích thước của cửa sổ Pygame là 1024x720 pixel bằng cách sử dụng các hằng số **SCREEN_WIDTH** và **SCREEN_HEIGHT**. Sau đó, nó tạo cửa sổ Pygame với kích thước đã được đặt và đặt tiêu đề của cửa sổ là "Main Menu" bằng cách sử dụng phương thức **pygame.display.set_caption().**

Hàm **pygame.display.set_caption()** trong Pygame được sử dụng để đặt tiêu đề của cửa sổ hiển thị chính trong ứng dụng hoặc trò chơi của bạn. Khi bạn chạy một ứng dụng Pygame, một cửa sổ sẽ xuất hiện trên màn hình, và **set_caption()** cho phép bạn đặt tên cho cửa sổ này.

```
background_image = pygame.image.load('background.jpg').convert()
background_rect = background_image.get_rect()
```

- Đoạn mã trên sử dụng Pygame để tải một hình ảnh nền từ tệp "background.jpg" và chuyển đổi nó thành một đối tượng hình ảnh Pygame. Sau đó, nó lấy hình chữ nhật chứa hình ảnh đó.
- **pygame.image.load('background.jpg'):** Hàm này tải hình ảnh từ tệp 'background.jpg' vào một đối tượng hình ảnh Pygame.
- **.convert():** Phương thức này chuyển đổi hình ảnh sang định dạng nhanh nhất cho hệ thống Pygame hiện tại, giúp tăng hiệu suất hiển thị.
- background_image.get_rect(): Phương thức này trả về hình chữ nhật chứa hình ảnh. Hình chữ nhật này có kích thước và vị trí phù hợp với hình ảnh, cho phép bạn dễ dàng vẽ hình ảnh lên màn hình Pygame.

Khi bạn sử dụng **background_rect,** bạn có thể dễ dàng điều chỉnh vị trí và kích thước của hình ảnh nền trên màn hình Pygame.

```
button_height = 50
spacing = 10  # Khoảng cách giữa các nút
total_button_height = button_height * 3 + spacing * 2
start_y = (SCREEN_HEIGHT - total_button_height) // 2  # Vị trí y bắt đầu để các nút nằm ở giữa

button1 = Button((SCREEN_WIDTH - 200) // 2, start_y, 200, 50, (0, 255, 0), 'Start Game')
button2 = Button((SCREEN_WIDTH - 200) // 2, start_y + button_height + spacing, 200, 50, (0, 255, 0), 'Options')
button3 = Button((SCREEN_WIDTH - 200) // 2, start_y + (button_height + spacing) * 2, 200, 50, (0, 255, 0), 'Quit')
```

Tạo ra ba đối tượng nút (button1, button2, button3) và đặt chúng ở vị trí dọc trên màn hình Pygame.

- **button_height = 50**: Đặt chiều cao cho mỗi nút là 50 pixel.
- **spacing = 10:** Đặt khoảng cách giữa các nút là 10 pixel.
- **total_button_height = button_height * 3 + spacing * 2**: Tính tổng chiều cao của tất cả các nút cộng với các khoảng cách giữa chúng. Trong trường hợp này, có ba nút nên tổng chiều cao là 50 * 3 + 10 * 2 = 180 pixel.
- **start_y = (SCREEN_HEIGHT - total_button_height) // 2**: Tính toán vị trí y bắt đầu để các nút nằm ở giữa theo chiều dọc của màn hình. Bạn lấy chiều cao của màn hình (**SCREEN_HEIGHT**), trừ đi tổng chiều cao của các nút và khoảng cách giữa chúng (**total_button_height**), sau đó chia cho 2 để đảm bảo các nút được căn giữa màn hình.
- **button1, button2, button3**: Tạo ba đối tượng nút với các tham số tương ứng là vị trí x, vị trí y, chiều rộng, chiều cao, màu sắc và văn bản của mỗi nút. Các nút được đặt ở cùng một vị trí x (vị trí ngang), nhưng vị trí y (vị trí dọc) tăng dần để chúng được đặt xen kẽ và căn giữa theo chiều dọc.

Cách thức căn chỉnh:

- Dòng 1**: start_y, 200, 50, (0, 255, 0),**
  - Đây là dòng tiếp theo của mã, trong đó **start_y** là vị trí y bắt đầu để đặt các nút.
  - **200** và **50** là chiều rộng và chiều cao của nút.
  - **(0, 255, 0)** là màu sắc của nút, trong trường hợp này có vẻ như màu xanh lá cây.
- Dòng 2: **start_y + button_height + spacing, 200, 50, (0, 255, 0), 'Options'**
  - Đây là dòng tiếp theo, đại diện cho nút thứ hai.
  - **start_y** + **button_height** + **spacing** là vị trí y của nút thứ hai, được tính bằng cách cộng **start_y** với chiều cao của nút và khoảng cách giữa các nút.
  - **200** và **b** là kích thước của nút.
  - **(0, 255, 0)** là màu sắc của nút (xanh lá cây).
  - **'Options'** là văn bản hiển thị trên nút này.
- Dòng 3: **start_y + (button_height + spacing) * 2, 200, 50, (0, 255, 0), 'Quit')**
  - Đây là dòng cuối cùng của đoạn mã, đại diện cho nút thứ ba.

- **start_y + (button_height + spacing) * 2** là vị trí y của nút thứ ba, được tính bằng cách cộng **start_y** với hai lần chiều cao của nút và một lần khoảng cách giữa các nút.
- **200** và **50** là kích thước của nút.
- **(0, 255, 0)** là màu sắc của nút (xanh lá cây).
- **'Quit'** là văn bản hiển thị trên nút này.

```
running = True
while running:
    # Hiển thị background image
    screen.blit(background_image, background_rect)

    # Vẽ các nút
    button1.draw(screen)
    button2.draw(screen)
    button3.draw(screen)
```

Trong đoạn mã trên, vòng lặp **while** đảm bảo rằng ứng dụng của bạn tiếp tục chạy cho đến khi biến **running** được đặt thành **False**. Bên trong vòng lặp, các bước sau được thực hiện:

- **screen.blit(background_image, background_rect):** Dòng này vẽ hình ảnh nền lên màn hình Pygame**. background_image** là hình ảnh nền đã được tải và **background_rect** là hình chữ nhật chứa hình ảnh. Hình ảnh nền sẽ được vẽ lên màn hình tại vị trí và kích thước được xác định bởi **background_rect.**
- **button1.draw(screen), button2.draw(screen), button3.draw(screen):** Các dòng này gọi phương thức **draw** của từng đối tượng nút để vẽ chúng lên màn hình Pygame. Các đối tượng nút **button1, button2, và button3** đã được tạo trước đó và đã có phương thức **draw** để vẽ chúng lên màn hình. Đối số **screen** cho biết rằng các nút sẽ được vẽ lên màn hình **screen**.

### c. Tạo event cho menu.

```
class OptionsScreen:
    def __init__(self):
        self.running = True

    def run(self):
        while self.running:
            # Vẽ các phần tử trên màn hình tùy chọn
            # Xử lý sự kiện cho màn hình tùy chọn
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.running = False
            pygame.display.update()
```

Lớp **OptionsScreen** này được thiết kế để quản lý màn hình tùy chọn trong ứng dụng của bạn. Dưới đây là cách nó hoạt động:

- **__init__:**
- Phương thức khởi tạo của lớp này chỉ đơn giản gán giá trị True cho thuộc tính **running**. Điều này cho biết rằng màn hình tùy chọn đang chạy.
- **run:**
- Phương thức này chứa vòng lặp chính của màn hình tùy chọn.
- Trong vòng lặp này, chương trình sẽ liên tục vẽ các phần tử trên màn hình tùy chọn và xử lý các sự kiện.
- Trong vòng lặp xử lý sự kiện **(for event in pygame.event.get()),** nếu sự kiện là **pygame.QUIT** (người dùng nhấn vào nút thoát), thuộc tính running sẽ được đặt thành False, từ đó thoát khỏi vòng lặp chính của màn hình tùy chọn.
- Cuối cùng, **pygame.display.update()** được gọi để cập nhật màn hình với các thay đổi mới.

Với cách thiết kế này, bạn có thể sử dụng lớp **OptionsScreen** để quản lý màn hình tùy chọn trong ứng dụng của mình, và điều chỉnh hành vi hoặc thêm các phần tử tùy chọn một cách linh hoạt bằng cách chỉnh sửa phương thức **run**.

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            pygame.quit()
            sys.exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            pos = pygame.mouse.get_pos()
            if button1.is_clicked(pos):
                print('Start Game clicked')
                # Xử lý khi nút "Start Game" được nhấp vào
                subprocess.Popen(['python', 'game.py'])
                pygame.quit()  # Tắt pygame
                sys.exit()
            elif button2.is_clicked(pos):
                print('Options clicked')
                # Xử lý khi nút "Options" được nhấp vào
            elif button3.is_clicked(pos):
                print('Quit clicked')
                running = False
                pygame.quit()
                sys.exit()
```

Đoạn mã trên là một phần của vòng lặp xử lý sự kiện trong ứng dụng Pygame của bạn. Dưới đây là cách nó hoạt động:

- **for event in pygame.event.get():** Vòng lặp này lặp qua tất cả các sự kiện trong hàng đợi sự kiện của Pygame.
- **if event.type == pygame.QUIT:** Nếu sự kiện là người dùng đóng cửa sổ, chương trình sẽ dừng lại.
- **pygame.quit():** Đóng cửa sổ Pygame.
- **sys.exit():** Thoát khỏi chương trình.
- **elif event.type == pygame.MOUSEBUTTONDOWN:**: Nếu sự kiện là người dùng nhấn chuột.
- **pos = pygame.mouse.get_pos():** Lấy tọa độ của chuột khi được nhấn.

- Nếu nút "**Start Game**" được nhấn:
  - In ra thông báo "**Start Game clicked**".

- Gọi **subprocess.Popen(['python', 'game.py'])** để chạy một tiến trình Python mới với tệp 'game.py'.
- **pygame.quit()** và **sys.exit()** để tắt Pygame và thoát khỏi chương trình.

- Nếu nút "**Options**" được nhấn:
  - In ra thông báo "**Options clicked**".
  - Xử lý khi nút "**Options**" được nhấp vào.

- Nếu nút "**Quit**" được nhấn:
  - In ra thông báo "**Quit clicked**".
  - Chương trình sẽ dừng lại, tắt Pygame và thoát khỏi chương trình.

```
    pygame.display.update()
```

- **pygame.display.update():** Cập nhật màn hình với các thay đổi mới sau khi xử lý sự kiện.

## 3. Game.py

### a. Import

Câu lệnh import trong Python là cách để chúng ta đưa các định nghĩa từ một module vào một chương trình Python. Khi bạn nhập một module hoặc lớp từ một script, bạn đang sử dụng các định nghĩa đã được định nghĩa trong đó để sử dụng trong chương trình của mình.

```
from scripts.spark import Spark
from scripts.utils import load_image, load_images, Animation
from scripts.entities import PhysicsEntity, Player, Enemy
from scripts.tilemap import Tilemap
from scripts.particle import Particle
```

- **from scripts.spark import Spark**: Câu lệnh này nhập lớp hoặc đối tượng **Spark** từ module **spark** trong gói **scripts**. Điều này có nghĩa là trong tệp **spark.py** bên trong thư mục **scripts**, bạn đã định nghĩa một lớp hoặc đối tượng có tên là **Spark**, và bạn muốn sử dụng nó trong mã của mình.
- **from scripts.utils import load_image, load_images, Animation:** Câu lệnh này nhập các hàm cụ thể **(load_image, load_images)** và một lớp (**Animation**) từ module **utils** trong gói **scripts**. Điều này có nghĩa là trong tệp utils.py bên trong thư mục **scripts**, bạn đã định nghĩa các hàm và lớp này và bạn muốn sử dụng chúng trong mã của mình.

- **from scripts.entities import PhysicsEntity, Player, Enemy**: Câu lệnh này nhập các lớp (**PhysicsEntity, Player, Enemy**) từ module **entities** trong gói **scripts**. Điều này có nghĩa là trong tệp **entities.py** bên trong thư mục **scripts**, bạn đã định nghĩa các lớp này và bạn muốn sử dụng chúng trong mã của mình.
- **from scripts.tilemap import Tilemap**: Câu lệnh này nhập lớp **Tilemap** từ module **tilemap** trong gói **scripts**. Điều này có nghĩa là trong tệp **tilemap.py** bên trong thư mục scripts, bạn đã định nghĩa lớp **Tilemap** và bạn muốn sử dụng nó trong mã của mình.
- **from scripts.particle import Particle:** Câu lệnh này nhập lớp **Particle** từ module **particle** trong gói scripts. Điều này có nghĩa là trong tệp **particle.py** bên trong thư mục **scripts**, bạn đã định nghĩa lớp **Particle** và bạn muốn sử dụng nó trong mã của mình.

Những câu lệnh import này giúp bạn tái sử dụng mã một cách hiệu quả bằng cách sử dụng các chức năng và lớp đã được định nghĩa trước đó trong các tệp riêng biệt.

### b. Assets (Tải Tài Nguyên)

```
self.assets = {
    'decor': load_images('tiles/decor'),
    'grass': load_images('grass'),
    'large_decor': load_images('large_decor'),
    'stone': load_images('stone'),
    'player': load_image('entities/player.png'),
    'background': load_image('background.png'),
    'enemy/idle': Animation(load_images('entities/enemy/idle'), img_dur=6),
    'enemy/run': Animation(load_images('entities/enemy/run'), img_dur=4),
    'player/idle' : Animation(load_images('entities/player/idle'), img_dur=6),
    'player/run' : Animation(load_images('entities/player/run'), img_dur=4),
    'player/jump' : Animation(load_images('entities/player/jump')),
    'player/slide' : Animation(load_images('entities/player/slide')),
    'player/wall_slide' : Animation(load_images('entities/player/wall_slide')),
    'particle/particle': Animation(load_images('particles/particle'), img_dur=6, loop=False),
    'projectile': load_image('projectile.png'),
}
```

- **'decor': load_images('tiles/decor'):** Tải các hình ảnh của các trang trí từ thư mục **'tiles/decor'** và lưu chúng trong một danh sách hình ảnh.
- **'grass': load_images('grass'):** Tải các hình ảnh của cỏ từ thư mục **'grass'** và lưu chúng trong một danh sách hình ảnh.
- **'large_decor': load_images('large_decor'):** Tải các hình ảnh của các trang trí lớn từ thư mục **'large_decor'** và lưu chúng trong một danh sách hình ảnh.
- **'stone': load_images('stone'):** Tải các hình ảnh của đá từ thư mục **'stone'** và lưu chúng trong một danh sách hình ảnh.

- **'player': load_image('entities/player.png'):** Tải hình ảnh của người chơi từ tệp **'entities/player.png'** và lưu nó.
- **'background': load_image('background.png'):** Tải hình ảnh của nền từ tệp **'background.png'** và lưu nó.
- **'enemy/idle', 'enemy/run'**: Tạo các hoạt ảnh của kẻ địch khi đứng yên và khi chạy bằng cách tải các hình ảnh tương ứng từ các thư mục **'entities/enemy/idle'** và **'entities/enemy/run'** và lưu chúng trong các đối tượng Animation.
- **'player/idle', 'player/run', 'player/jump', 'player/slide', 'player/wall_slide'**: Tương tự như trên, tạo các hoạt ảnh của người chơi ở trạng thái đứng yên, chạy, nhảy, trượt và trượt trên tường từ các tệp hình ảnh tương ứng.
- **'particle/particle':** Tạo một hiệu ứng hạt bằng cách tải các hình ảnh từ thư mục **'particles/particle'** và lưu chúng trong một đối tượng Animation không lặp lại.
- **'projectile': load_image('projectile.png'):** Tải hình ảnh của dự đoán từ tệp **'projectile.png'** và lưu nó.

```
self.player = Player(self, (50, 50), (8,15))

self.tilemap = Tilemap(self, tile_size= 16)

self.level = 0
self.load_level(self.level)
```

- **self.player = Player(self, (50, 50), (8,15))**: Tạo một đối tượng người chơi (hoặc nhân vật chính) với vị trí ban đầu là **(50, 50)** trên bản đồ và kích thước là **(8, 15).** Đối tượng người chơi này được lưu vào biến **self.player.**
- **self.tilemap = Tilemap(self, tile_size= 16):** Tạo một đối tượng bản đồ tile với kích thước tile là 16x16. Đối tượng bản đồ tile này được lưu vào biến **self.tilemap**.
- **self.level = 0**: Khởi tạo biến **self.level** với giá trị ban đầu là 0, có thể đại diện cho cấp độ hoặc màn chơi hiện tại.
- **self.load_level(self.level):** Gọi phương thức **load_level()** với tham số **self.level**, có thể để tải cấp độ hoặc màn chơi cụ thể được định nghĩa trong trò chơi của bạn. Điều này có thể là phần tiếp theo của quy trình khởi tạo cài đặt cấp độ hoặc màn chơi ban đầu cho trò chơi của bạn.

### c. Load Level

```
def load_level(self, map_id):
    self.tilemap.load(f'data/maps/{map_id}.json')
    self.enemies = []
    for spawner in self.tilemap.extract([('spawners', 0), ('spawners', 1)]):
        if spawner['variant'] == 0:
            self.player.pos = spawner['pos']
        else:
            self.enemies.append(Enemy(self, spawner['pos'], (8, 15)))


    self.particles = []
    self.projectiles = []
    self.sparks = []

    self.scroll = [0, 0] #camera pos in top left
    self.dead = 0

    self.transition = -30
```

- **self.tilemap.load(f'data/maps/{map_id}.json'):** Tải bản đồ tile từ tệp JSON tương ứng với map_id được cung cấp vào **self.tilemap**. Tệp JSON này thường chứa thông tin về bản đồ tile, bao gồm vị trí của các tile và các đối tượng khác trên bản đồ.
- **self.enemies = []:** Khởi tạo một danh sách rỗng để lưu trữ các đối tượng kẻ địch trong cấp độ.
- **for spawner in self.tilemap.extract([('spawners', 0), ('spawners', 1)]):**: Duyệt qua tất cả các đối tượng 'spawner' trên bản đồ và trích xuất thông tin về chúng.
  - Nếu **spawner['variant']** là 0, nghĩa là đây là vị trí xuất phát của người chơi, nên cập nhật vị trí của người chơi thành **spawner['pos'].**
  - Nếu không, đây là vị trí xuất hiện của một kẻ địch, vì vậy tạo một đối tượng Enemy mới với vị trí **spawner['pos']** và kích thước **(8, 15),** sau đó thêm vào danh sách **self.enemies.**
- **self.particles = [], self.projectiles = [], self.sparks = []:** Khởi tạo các danh sách rỗng để lưu trữ các hạt, dự đoán và tia sét trong cấp độ. Các đối tượng này có thể được tạo ra và quản lý trong quá trình chơi.
- **self.scroll = [0, 0]:** Đặt vị trí cuộn (**scroll**) về [0, 0], có thể là vị trí camera ban đầu trên bản đồ.
- **self.dead = 0:** Khởi tạo biến **self.dead** với giá trị ban đầu là 0, có thể đại diện cho số lượng kẻ địch đã bị tiêu diệt trong cấp độ.
- **self.transition = -30**: Đặt giá trị **self.transition** về -30, có thể là một biến được sử dụng để kiểm soát các hiệu ứng chuyển tiếp khi chuyển sang cấp độ mới trong trò chơi.

### d. Phương thức chính, Trung tâm trò chơi

Phương thức **run()** chính là trung tâm của vòng lặp chính của trò chơi. Nó điều khiển toàn bộ luồng logic và hiển thị của trò chơi, từ xử lý sự kiện đến vẽ đồ họa trên màn hình. Dưới đây là một phân tích chi tiết về hoạt động của phương thức này:

- **Hiển thị nền**: Đầu tiên, phương thức này vẽ nền của trò chơi lên màn hình. Nền thường là hình ảnh hoặc một bản đồ màu tĩnh.
- **Quản lý cấp độ**: Phương thức kiểm tra xem tất cả kẻ địch đã bị tiêu diệt chưa. Nếu tất cả kẻ địch đã bị tiêu diệt, trò chơi sẽ chuyển sang cấp độ tiếp theo.
- **Xử lý chuyển tiếp**: Trong quá trình chuyển đổi giữa các cấp độ hoặc trạng thái trong trò chơi, phương thức quản lý các hiệu ứng chuyển tiếp, như làm đen màn hình hoặc hiển thị hình ảnh chuyển tiếp.
- **Di chuyển camera**: Camera của trò chơi được điều chỉnh để theo dõi người chơi hoặc các đối tượng khác trên màn hình.
- **Vẽ bản đồ tile**: Phương thức vẽ bản đồ tile lên màn hình, bao gồm cả các đối tượng và nền của trò chơi.
- **Quản lý và vẽ kẻ địch**: Tất cả các kẻ địch trong trò chơi được cập nhật và vẽ lên màn hình. Những kẻ địch đã bị tiêu diệt sẽ được loại bỏ khỏi danh sách.
- **Quản lý và vẽ người chơi**: Người chơi được cập nhật và vẽ lên màn hình, dựa trên các thao tác của người chơi và trạng thái của bản đồ.
- **Quản lý và vẽ các dự đoán (projectile):** Các dự đoán như đạn, tia laser, hoặc các vật thể di động khác được cập nhật và vẽ lên màn hình. Các dự đoán va chạm với các vật thể khác sẽ gây ra các hiệu ứng phụ phức tạp.
- **Quản lý và vẽ hiệu ứng (sparks, particles):** Các hiệu ứng như tia sét, hạt bụi, hoặc các hiệu ứng khác được cập nhật và vẽ lên màn hình, thường là để tạo ra cảm giác sống động và thú vị cho trò chơi.
- **Xử lý sự kiện từ bàn phím**: Phương thức này lắng nghe và xử lý các sự kiện từ bàn phím, như di chuyển của người chơi hoặc các thao tác đặc biệt.
- **Vẽ tất cả lên màn hình và cập nhật màn hình**: Cuối cùng, tất cả các thay đổi được vẽ lên màn hình và màn hình được cập nhật để hiển thị những thay đổi mới nhất. Đảm bảo rằng trò chơi chạy ở tốc độ ổn định là mục tiêu cuối cùng của phương thức **run().**

```
self.display.blit(self.assets['background'], (0, 0))

if not len(self.enemies):
    self.transition += 1
    if self.transition > 30:
        self.level = min(self.level + 1, len(os.listdir('data/maps'))-1)
        self.load_level(self.level)
if self.transition < 0:
    self.transition += 1

if self.dead:
    self.dead += 1
    if self.dead >= 10:
        self.transition = min(self.transition + 1,30)
    if self.dead > 40:
        self.load_level(self.level)

self.scroll[0] += (self.player.rect().centerx - self.display.get_width() / 2 - self.scroll[0]) / 60
self.scroll[1] += (self.player.rect().centery - self.display.get_height() / 2 - self.scroll[1]) / 60
render_scroll = (int(self.scroll[0]), int(self.scroll[1]))

self.tilemap.render(self.display, offset = render_scroll)
```

- Vòng lặp **while True** cho biết đây là vòng lặp trò chơi chính, chạy vô hạn cho đến khi chương trình được thoát.
- **self.display.blit(self.assets['background'], (0, 0)):** Dòng này blits (vẽ) hình nền lên bề mặt hiển thị tại tọa độ (0, 0). Điều này hiệu quả làm sạch màn hình bằng hình nền trước khi vẽ các yếu tố khác.
- **Phần với if not len(self.enemies):** kiểm tra xem có bất kỳ kẻ địch nào còn lại không. Nếu không, nó khởi đầu một giai đoạn chuyển tiếp. Trong thời gian này, nó kiểm tra xem thời gian chuyển đổi đã vượt quá 30 khung hình chưa. Nếu có, nó tải cấp độ tiếp theo, nếu có sẵn.
- **if self.transition < 0::** Khối này đảm bảo rằng giá trị chuyển đổi không thấp hơn 0.
- **if self.dead::** Khối này kiểm tra xem người chơi có chết không. Nếu người chơi đã chết, nó tăng biến 'dead'. Nếu người chơi đã chết hơn 10 khung hình, nó khởi đầu một chuyển đổi. Sau 40 khung hình bị chết, nó tải lại cấp độ hiện tại.
- Các dòng tiếp theo tính toán cuộn màn hình để theo dõi vị trí của người chơi. Nó cập nhật biến **scroll** để mượt mà theo dõi trung tâm của người chơi trên màn hình.
- **render_scroll = (int(self.scroll[0]), int(self.scroll[1])):** Dòng này chuyển đổi các giá trị cuộn dạng số thực thành số nguyên, sau đó được sử dụng để vẽ các đối tượng trò chơi liên quan đến vị trí cuộn màn hình.

Dòng này tính toán di chuyển màn hình để theo dõi vị trí của người chơi. Đây là một phần quan trọng trong việc tạo ra hiệu ứng mượt mà khi người chơi di chuyển qua lại trên bản đồ.

- **self.scroll[0] += (self.player.rect().centerx - self.display.get_width() / 2 - self.scroll[0]) / 60**: Đoạn này tính toán phần tử thứ nhất của **self.scroll**, nói cách khác, là di chuyển theo chiều ngang của màn hình. Cụ thể:
- **self.player.rect().centerx** là tọa độ x của trung điểm của người chơi trên màn hình.
- **self.display.get_width() / 2** là nửa chiều rộng của màn hình, đại diện cho tọa độ x của trung điểm của màn hình.
- **self.scroll[0]** là vị trí hiện tại của màn hình theo chiều ngang.
- Công thức này tính toán sự chênh lệch giữa vị trí của người chơi và trung điểm của màn hình, sau đó chia cho một hằng số (60 trong trường hợp này) để tạo ra hiệu ứng di chuyển mượt mà. Kết quả được cộng vào vị trí hiện tại của màn hình.
- **self.scroll[1] += (self.player.rect().centery - self.display.get_height() / 2 - self.scroll[1]) / 60**: Tương tự như trên, dòng này tính toán di chuyển theo chiều dọc của màn hình. Cụ thể:
- **self.player.rect().centery** là tọa độ y của trung điểm của người chơi trên màn hình.
- **self.display.get_height() / 2** là nửa chiều cao của màn hình, đại diện cho tọa độ y của trung điểm của màn hình.
- **self.scroll[1]** là vị trí hiện tại của màn hình theo chiều dọc.
- Công thức tương tự tính toán sự chênh lệch giữa vị trí của người chơi và trung điểm của màn hình, sau đó chia cho một hằng số (60 trong trường hợp này) để tạo ra hiệu ứng di chuyển mượt mà. Kết quả được cộng vào vị trí hiện tại của màn hình.

```
self.tilemap.render(self.display, offset = render_scroll)
```

Dòng này gọi phương thức render của đối tượng self.tilemap, để vẽ bản đồ lên màn hình.

- **self.tilemap:** Đây là đối tượng bản đồ (tilemap) của trò chơi.
- **.render**(: Gọi phương thức render của đối tượng **self.tilemap**.
- **self.display:** Tham số đầu tiên của phương thức **render** là **self.display,** đại diện cho màn hình hiển thị trò chơi. Bản đồ sẽ được vẽ lên màn hình này.
- **offset = render_scroll:** Tham số **offset** cho phép điều chỉnh vị trí của bản đồ trên màn hình. Giá trị của **offset** được tính từ biến **render_scroll**, giúp cho bản đồ di chuyển đồng bộ với vị trí của người chơi trên màn hình.

Khi phương thức render này được gọi, bản đồ của trò chơi sẽ được vẽ lên màn hình tại vị trí được xác định bởi render_scroll, điều này giúp người chơi có thể nhìn thấy một phần của bản đồ tương ứng với vị trí của họ.

❖ **Loại bỏ kẻ địch khi giết**

```
for enemy in self.enemies.copy():
    kill = enemy.update(self.tilemap, (0, 0))
    enemy.render(self.display, offset=render_scroll)
    if kill:
        self.enemies.remove(enemy)
```

Dòng mã này lặp qua danh sách các kẻ địch trong self.enemies (hoặc một bản sao của nó để tránh thay đổi danh sách trong quá trình lặp), cập nhật chúng và vẽ chúng lên màn hình. Nếu một kẻ địch được hủy diệt trong quá trình cập nhật, nó sẽ bị loại bỏ khỏi danh sách.

- **for enemy in self.enemies.copy()::** Vòng lặp này lặp qua từng đối tượng kẻ địch trong danh sách **self.enemies.** Thông qua việc sử dụng **copy()** để tạo một bản sao của danh sách **self.enemies**, chúng ta đảm bảo rằng chúng ta có thể loại bỏ các kẻ địch trong quá trình lặp mà không ảnh hưởng đến vòng lặp.
- **kill = enemy.update(self.tilemap, (0, 0)):** Phương thức **update** được gọi trên mỗi đối tượng kẻ địch để cập nhật trạng thái của chúng. Thông thường, phương thức **update** sẽ trả về một giá trị boolean để chỉ ra liệu kẻ địch đã bị hủy diệt hay không.
- **enemy.render(self.display, offset=render_scroll):** Sau khi cập nhật, kẻ địch được vẽ lên màn hình bằng cách gọi phương thức **render**, với màn hình **self.display** và **offset** được truyền vào để xác định vị trí của kẻ địch trên màn hình.
- **if kill: self.enemies.remove(enemy):** Nếu biến **kill** là True, ngụ ý rằng kẻ địch đã bị hủy diệt. Trong trường hợp này, kẻ địch được loại bỏ khỏi danh sách **self.enemies** để không còn được cập nhật hoặc vẽ lên màn hình trong các lần lặp tiếp theo.

❖ **Kiểm tra người chơi còn sống hay không?**

```
if not self.dead:

    self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0)) # chỉ có 3 kết quả: -1, 0, 1
    self.player.render(self.display, offset = render_scroll)
```

Dòng mã này kiểm tra xem người chơi có sống hay không, và nếu có, cập nhật và vẽ người chơi lên màn hình.

- **if not self.dead::** Điều kiện này kiểm tra xem người chơi có còn sống không. Nếu **self.dead** bằng False, tức là người chơi vẫn còn sống.

- **self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0)):** Sau khi kiểm tra xem người chơi có sống không, phương thức **update** được gọi trên đối tượng người chơi để cập nhật trạng thái của họ. Tham số **self.tilemap** và **(self.movement[1] - self.movement[0], 0)** được truyền vào, chúng được sử dụng để xác định các di chuyển mới của người chơi trên bản đồ.
- **self.player.render(self.display, offset=render_scroll):** Sau khi cập nhật, người chơi được vẽ lên màn hình bằng cách gọi phương thức **render**, với màn hình **self.display** và **offset** được truyền vào để xác định vị trí của người chơi trên màn hình.

### ❖ Xử lý di chuyển

```
for projectile in self.projectiles.copy():
    projectile[0][0] += projectile[1]
    projectile[2] += 1
    img = self.assets['projectile']
    self.display.blit(img, (projectile[0][0] - img.get_width() / 2 - render_scroll[0], projectile[0][1] - img.get_height() / 2 - render_scroll[1]))
    if self.tilemap.solid_check(projectile[0]):
        self.projectiles.remove(projectile)
        for i in range(4):
            # self.sparks.append(Spark(self.projectiles[0] + (math.pi if projectile[1] > 0 else 0), random.random() - 0.5, 2 + random.random()))
            self.sparks.append(Spark(projectile[0] , random.random() - 0.5, 2 + random.random()))
    elif projectile[2] > 360:
        self.projectiles.remove(projectile)
    elif abs(self.player.dashing) < 50:
        if self.player.rect().collidepoint(projectile[0]):
            self.projectiles.remove(projectile)
            self.dead += 1
            for i in range(30):
                angle = random.random() * math.pi * 2
                speed = random.random() * 5
                self.sparks.append(Spark(self.player.rect().center, angle, 2 + random.random()))
                self.particles.append(Particle(self, 'particle', self.player.rect().center, velocity=[math.cos(angle + math.pi) * speed * 0.5, math.sin(angle + m
```

- **for projectile in self.projectiles.copy()::** Duyệt qua mỗi viên đạn trong danh sách các viên đạn của đối tượng **self**, sử dụng một bản sao của danh sách.
- **projectile[0][0] += projectile[1]:** Cập nhật tọa độ x của viên đạn bằng cách thêm vận tốc của nó.
- **projectile[2] += 1**: Tăng biến đếm của viên đạn. Có thể là một biến đếm thời gian hoặc số lần viên đạn đã được cập nhật.
- **img = self.assets['projectile']:** Lấy hình ảnh của viên đạn từ tài nguyên của đối tượng self.
  - **self.display.blit(img, (projectile[0][0] - img.get_width() / 2 - render_scroll[0], projectile[0][1] - img.get_height() / 2 - render_scroll[1])):** Vẽ hình ảnh của viên đạn lên màn hình tại vị trí hiện tại của nó trừ đi nửa chiều rộng và cao của hình ảnh, cộng với giá trị cuộn (scroll) của màn hình.
  - **self.display.blit(img, ...):** Đây là một phương thức để vẽ hình ảnh lên một bề mặt cụ thể (trong trường hợp này là self.display).
  - **img** là hình ảnh của viên đạn, đã được lấy từ các tài nguyên của trò chơi.

- **(projectile[0][0] - img.get_width() / 2 - render_scroll[0], projectile[0][1] - img.get_height() / 2 - render_scroll[1]):** Đây là tọa độ vị trí mà viên đạn sẽ được vẽ lên. Hãy xem xét các phép tính:
- **projectile[0][0] - img.get_width() / 2:** Tính toán tọa độ x mà viên đạn sẽ được vẽ, sao cho viên đạn được vẽ tại giữa theo chiều ngang**. projectile[0][0]** là tọa độ x hiện tại của viên đạn**, img.get_width() / 2** là nửa chiều rộng của hình ảnh, điều này đảm bảo rằng viên đạn sẽ được vẽ từ trung tâm của nó.
- **projectile[0][1] - img.get_height() / 2:** Tính toán tọa độ y mà viên đạn sẽ được vẽ, sao cho viên đạn được vẽ tại giữa theo chiều dọc. Tương tự như trên, **projectile[0][1]** là tọa độ y hiện tại của viên đạn, **img.get_height() / 2** là nửa chiều cao của hình ảnh.
- **- render_scroll[0] và - render_scroll[1]:** Điều này có thể liên quan đến việc di chuyển camera trong trò chơi. Nếu **render_scroll** không đổi, nó sẽ không ảnh hưởng đến vị trí của viên đạn trên màn hình, nhưng nếu nó thay đổi, viên đạn sẽ di chuyển tương ứng để giữ cho việc vẽ trên màn hình là đúng.

- **if self.tilemap.solid_check(projectile[0]):**: Kiểm tra xem viên đạn có va chạm với các vật cản không bằng cách sử dụng phương thức **solid_check** của self.tilemap.
- **self.projectiles.remove(projectile):** Nếu viên đạn va chạm, loại bỏ nó khỏi danh sách viên đạn.
- **for i in range(4):**: Với mỗi lần lặp, thêm 4 phát lửa nhỏ (sparks) mới vào danh sách **self.sparks.**
  - **self.sparks.append(Spark(projectile[0], random.random() - 0.5, 2 + random.random())):** Tạo một đối tượng Spark mới tại vị trí của viên đạn va chạm với vận tốc và hướng ngẫu nhiên, sau đó thêm nó vào danh sách self.sparks.
  - **self.sparks.append(...):** Đây là cách thêm một phần tử mới vào danh sách **self.sparks**. Đối tượng **"spark"** mới được tạo sẽ được thêm vào cuối danh sách.
  - **Spark(projectile[0], random.random() - 0.5, 2 + random.random()):** Đây là cách tạo một đối tượng "**spark**" mới. Cụ thể:
  - **Spark(...)** là cách tạo một thể hiện mới của lớp Spark.
  - **projectile[0]** là tọa độ ban đầu của viên đạn mà spark được tạo ra từ. Thường là tọa độ của viên đạn đã va chạm với một vật thể.
  - **random.random() - 0.5:** Đoạn này tạo ra một giá trị ngẫu nhiên trong khoảng từ -0.5 đến 0.5. Điều này có thể được sử dụng để tạo ra một sự biến đổi ngẫu nhiên trong hướng di chuyển của spark.
  - **2 + random.random():** Đoạn này tạo ra một giá trị ngẫu nhiên trong khoảng từ 2 đến 3. Điều này có thể được sử dụng để tạo ra một biến đổi ngẫu nhiên trong tốc độ di chuyển của spark.

- **elif projectile[2] > 360**:: Điều kiện này kiểm tra nếu biến đếm của viên đạn **(projectile[2])** vượt quá giá trị 360. Nếu điều kiện này đúng, có thể đang xử lý việc loại bỏ viên đạn khỏi danh sách **self.projectiles**.
- **self.projectiles.remove(projectile):** Nếu điều kiện trên được thỏa mãn, viên đạn sẽ được loại bỏ khỏi danh sách **self.projectiles**.
- **elif abs(self.player.dashing) < 50::** Điều kiện này kiểm tra nếu giá trị tuyệt đối của trạng thái dashing của người chơi (**self.player.dashing**) nhỏ hơn 50.
- **if self.player.rect().collidepoint(projectile[0])::** Điều kiện này kiểm tra xem tọa độ hiện tại của viên đạn **(projectile[0])** có nằm trong phạm vi của hình chữ nhật đại diện cho người chơi không.
- **self.projectiles.remove(projectile):** Nếu điều kiện trên được thỏa mãn, viên đạn sẽ được loại bỏ khỏi danh sách **self.projectiles.**
- **self.dead += 1**: Biến **self.dead** được tăng lên một đơn vị. Đây có thể là biến để theo dõi số lượng đạn đã gây ra sát thương cho người chơi.
- **for i in range(30)::** Vòng lặp này tạo ra 30 hạt (particles) và 30 spark mới, có thể là hiệu ứng hoặc các phần tử vizual cho việc va chạm của người chơi với viên đạn.
- **angle = random.random() * math.pi * 2:** Tạo một góc ngẫu nhiên từ 0 đến 2π (360 độ).
- **speed = random.random() * 5:** Tạo một tốc độ ngẫu nhiên từ 0 đến 5.
  - **self.sparks.append(Spark(self.player.rect().center, angle, 2 + random.random())):** Thêm một spark mới vào danh sách self.sparks với vị trí ban đầu ở trung tâm của người chơi, góc và tốc độ ngẫu nhiên.
  - **self.particles.append(Particle(...)):** Thêm một hạt (particle) mới vào danh sách **self.particles** với các thông số như hình ảnh, vị trí ban đầu, vận tốc và khung hình ngẫu nhiên.

```
for spark in self.sparks.copy():
    kill = spark.update()
    spark.render(self.display, offset=render_scroll)
    if kill:
        self.sparks.remove(spark)
```

- **for spark in self.sparks.copy()::** Duyệt qua mỗi spark trong danh sách **self.sparks**. Việc sử dụng **self.sparks.copy()** thay vì **self.sparks** trực tiếp có thể liên quan đến việc loại bỏ các phần tử trong quá trình lặp.
- **kill = spark.update():** Gọi phương thức **update()** của spark để cập nhật trạng thái của nó và kiểm tra xem nó có cần phải bị loại bỏ không. Giá trị trả về **kill** có thể là **True** nếu spark cần được loại bỏ và **False** nếu không.

- **spark.render(self.display, offset=render_scroll):** Gọi phương thức **render()** của spark để vẽ nó lên màn hình. Tham số **self.display** là bề mặt mà spark sẽ được vẽ lên, và **offset=render_scroll** có thể là vị trí di chuyển của spark trên màn hình.
- **if kill: self.sparks.remove(spark):** Nếu biến **kill** được đặt thành **True**, spark sẽ được loại bỏ khỏi danh sách **self.sparks**. Điều này xảy ra nếu spark đã hoàn thành một chu trình hoạt động hoặc nếu nó đã đi qua ranh giới của màn hình.

```
for particle in self.particles.copy():
    kill = particle.update()
    particle.render(self.display, offset=render_scroll)
    # if particle.type == 'leaf':
    #     particle.pos[0] += math.sin(particle.animation.frame * 0.035) * 0.3 #0.035 la toc do cham thoi
    if kill:
        self.particles.remove(particle)
```

- **for particle in self.particles.copy()::** Duyệt qua mỗi particle trong danh sách **self.particles.**
- **kill = particle.update()**: Gọi phương thức **update()** của particle để cập nhật trạng thái của nó và kiểm tra xem nó có cần phải bị loại bỏ không. Giá trị trả về **kill** có thể là **True** nếu particle cần được loại bỏ và **False** nếu không.
- **particle.render(self.display, offset=render_scroll):** Gọi phương thức **render()** của particle để vẽ nó lên màn hình. Tham số **self.display** là bề mặt mà particle sẽ được vẽ lên, và **offset=render_scroll** có thể là vị trí di chuyển của particle trên màn hình.
- **if kill: self.particles.remove(particle):** Nếu biến **kill** được đặt thành **True**, particle sẽ được loại bỏ khỏi danh sách **self.particles**. Điều này xảy ra nếu particle đã hoàn thành một chu trình hoạt động hoặc nếu nó đã đi qua ranh giới của màn hình.

❖ **Xử lý sự kiện từ bàn phím**

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:   # Click x on window
        pygame.quit()
        sys.exit()
    if event.type == pygame.KEYDOWN:    # dang nhan nut
        if event.key == pygame.K_LEFT:
            self.movement[0] = True
        if event.key == pygame.K_RIGHT:
            self.movement[1] = True
        if event.key == pygame.K_UP:
            # self.player.velocity[1] = -3  # override van toc
            self.player.jump()
        if event.key == pygame.K_x:
            self.player.dash()
    if event.type == pygame.KEYUP:   # tha nut ra
        if event.key == pygame.K_LEFT:
            self.movement[0] = False
        if event.key == pygame.K_RIGHT:
            self.movement[1] = False
```

- **for event in pygame.event.get()::** Duyệt qua tất cả các sự kiện trong hàng đợi sự kiện của Pygame.
- **if event.type == pygame.QUIT**:: Kiểm tra xem sự kiện có phải là sự kiện thoát khỏi cửa sổ không. Nếu là, trò chơi sẽ được kết thúc bằng cách gọi **pygame.quit()** để dọn dẹp tài nguyên và **sys.exit()** để thoát khỏi chương trình.
- **if event.type == pygame.KEYDOWN::** Kiểm tra xem sự kiện có phải là sự kiện phím được nhấn không.
- **if event.key == pygame.K_LEFT::** Kiểm tra xem phím đã được nhấn có phải là phím mũi tên trái không. Nếu có, **self.movement[0] sẽ** được đặt thành True, cho biết rằng người chơi đang di chuyển sang trái.
- **if event.key == pygame.K_RIGHT::** Tương tự như trên, nhưng kiểm tra phím mũi tên phải và thiết lập self.movement[1] thành True để biểu thị người chơi đang di chuyển sang phải.
- **if event.key == pygame.K_UP:: Kiểm** tra xem phím đã được nhấn có phải là phím mũi tên lên không. Nếu có, gọi phương thức **jump()** của người chơi để thực hiện nhảy.
- **if event.key == pygame.K_x::** Kiểm tra xem phím đã được nhấn có phải là phím "X" không. Nếu có, gọi phương thức **dash()** của người chơi để thực hiện động tác "dash".

- **if event.type == pygame.KEYUP::** Kiểm tra xem sự kiện có phải là sự kiện phím được thả ra không.

Tương tự như các điều kiện ở trên, các điều kiện trong phần này kiểm tra xem phím đã được thả ra là phím nào và cập nhật trạng thái tương ứng trong self.movement cho phép người chơi ngừng di chuyển khi phím không còn được nhấn nữa.

❖ **Hiển thị trò chơi và khung hình**

```
self.screen.blit(pygame.transform.scale(self.display, self.screen.get_size()), (0, 0))
pygame.display.update() # Phải có
self.clock.tick(60)  # 60FPS
```

- **self.screen.blit(pygame.transform.scale(self.display, self.screen.get_size()), (0, 0)):** Dòng này thực hiện việc vẽ hình ảnh từ bề mặt self.display lên màn hình **self.screen.** Trước khi vẽ, hình ảnh trên **self.display** được co giãn hoặc thu nhỏ để phù hợp với kích thước của màn hình sử dụng pygame.transform.scale. Ở đây, (0, 0) là tọa độ góc trên bên trái của màn hình.
- **pygame.display.update():** Dòng này cập nhật nội dung trên màn hình, đảm bảo rằng bất kỳ thay đổi nào trong hình ảnh hiển thị được áp dụng.
- **self.clock.tick(60):** Dòng này giữ cho trò chơi chạy ở tốc độ cố định, ở đây là 60 khung hình mỗi giây. Điều này giúp duy trì sự liên tục và mịn màng của trò chơi.

## 4. tilemap.py

Dùng để đọc file json chứa các ô được tạo cho map và tạo ra bản đồ game play.

```
AUTOTILE_MAP = {
    tuple(sorted([(1, 0), (0, 1)])) : 0,
    tuple(sorted([(1, 0), (0, 1), (-1, 0)])) : 1,
    tuple(sorted([(-1, 0), (0, 1)])): 2,
    tuple(sorted([(-1, 0), (0, -1), (0, 1)])): 3,
    tuple(sorted([(-1, 0), (0, -1)])): 4,
    tuple(sorted([(-1, 0), (0, -1), (1, 0)])): 5,
    tuple(sorted([(1, 0), (0, -1)])): 6,
    tuple(sorted([(1, 0), (0, -1), (0, 1)])): 7,
    tuple(sorted([(1, 0), (-1, 0), (0, 1), (0, -1)])): 8,

}

NEIGHBOR_OFFSETS = [(-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (0, 0), (-1, 1), (0, 1), (1, 1)]
PHYSICS_TILES = {'grass', 'stone', }
AUTOTILE_TYPES = {'grass', 'stone'}
```

**AUTOTILE_MAP:** Đây là một từ điển dùng để ánh xạ các tập hợp các ô lân cận thành một giá trị duy nhất, được sử dụng để xác định loại tile autotile tương ứng. Cụ thể:

- Key: Mỗi key là một tuple các tọa độ (dx, dy) đại diện cho sự khác biệt giữa tọa độ của ô đang xét và các ô lân cận.
- Value: Giá trị của key tương ứng với loại tile autotile.

**NEIGHBOR_OFFSETS:** Đây là một danh sách các tọa độ (dx, dy) đại diện cho vị trí tương đối của các ô lân cận so với ô hiện tại. Cụ thể, các tọa độ này đại diện cho 8 ô lân cận xung quanh ô hiện tại.

**PHYSICS_TILES** và **AUTOTILE_TYPES**: Đây là hai tập hợp các chuỗi, đại diện cho các loại tile khác nhau. Cụ thể:

- **PHYSICS_TILES**: Là tập hợp các loại tile được sử dụng để xác định vật lý, chẳng hạn như 'grass' (cỏ) hoặc 'stone' (đá).
- **AUTOTILE_TYPES**: Là tập hợp các loại tile được sử dụng cho autotiling.

## a. Đặt giá trị 1 ô mặc định là 16x16

```
def __init__(self, game, tile_size = 16): #kích thước 1 ô mặc định là 16px
    self.game = game
    self.tile_size = tile_size
    self.tilemap = {}
    self.offgrid_tiles = []
```

Phương thức **__init__** này khởi tạo một đối tượng Tilemap với các thuộc tính cơ bản. Dưới đây là ý nghĩa của từng tham số:

- **game**: Tham số này là một tham chiếu đến đối tượng trò chơi chứa Tilemap. Việc này cho phép Tilemap truy cập các phương thức và thuộc tính của trò chơi khi cần thiết.
- **tile_size**: Đây là kích thước của mỗi ô tile trong tilemap, mặc định là 16 pixel.
- **tilemap**: Đây là một từ điển để lưu trữ dữ liệu về tilemap. Cụ thể, từ điển này sẽ ánh xạ từng cặp tọa độ (x, y) của ô tile tới loại tile tương ứng.
- **offgrid_tiles**: Đây là một danh sách để lưu trữ các ô tile không nằm trên lưới chính. Các ô tile này có thể nằm ngoài khu vực lưới hoặc không thuộc vào lưới chính.

Phương thức **extract** này được sử dụng để trích xuất các ô tile từ tilemap dựa trên một danh sách các cặp id (loại và biến thể) được chỉ định. Dưới đây là ý nghĩa của từng tham số và bước xử lý:

- **id_pairs**: Danh sách các cặp id (loại và biến thể) cần trích xuất từ tilemap.

- **keep**: Một cờ chỉ định liệu các ô tile trích xuất nên được giữ lại trong tilemap hay không. Mặc định là **False**, nghĩa là các ô tile được trích xuất sẽ bị xóa khỏi tilemap.
- **matches**: Danh sách các ô tile phù hợp với các cặp id được trích xuất.

Phương thức extract hoạt động như sau:

- **matches = []:** Khởi tạo danh sách trống để lưu trữ các ô tile phù hợp.
- Vòng lặp đầu tiên: Duyệt qua tất cả các ô tile không nằm trên lưới chính (**offgrid_tiles**).
  - Nếu id của ô tile khớp với bất kỳ cặp id nào trong **id_pairs**, ô tile đó được thêm vào danh sách **matches**.
  - Nếu không muốn giữ lại ô tile trong tilemap (**keep** là False), ô tile đó sẽ được loại bỏ khỏi danh sách **offgrid_tiles.**
- Vòng lặp thứ hai: Duyệt qua tất cả các ô tile trên lưới chính (**tilemap**).
  - Nếu id của ô tile khớp với bất kỳ cặp id nào trong **id_pairs**, ô tile đó được thêm vào danh sách **matches**.
  - Nếu không muốn giữ lại ô tile trong tilemap (**keep** là False), ô tile đó sẽ được loại bỏ khỏi tilemap.
- Cuối cùng, phương thức trả về danh sách **matches** chứa tất cả các ô tile phù hợp được trích xuất từ tilemap.

```
def tiles_around(self, pos): #bien pixel thanh o grid, tim tat ca cac tile xung quanh 1 vi tri
    tiles = []
    tile_loc = (int(pos[0] // self.tile_size), int(pos[1] // self.tile_size))
    #sử dụng phần nguyên sàn để đơn giản hóa tính toán vs nếu sử dụng phần nguyên trần thì tọa độ có thể nằm ngoài lưới grid
    for offset in NEIGHBOR_OFFSETS:
        check_loc = str(tile_loc[0] + offset[0]) + ';' + str(tile_loc[1] + offset[1])
        if check_loc in self.tilemap:
            tiles.append(self.tilemap[check_loc])
    return tiles
```

Phương thức **tiles_around** này được sử dụng để tìm tất cả các ô tile xung quanh một vị trí cụ thể trên lưới. Dưới đây là ý nghĩa của từng tham số và bước xử lý:

- **pos**: Vị trí pixel của ô tile trên lưới.
- **tiles**: Danh sách các ô tile xung quanh vị trí cụ thể.
- **tile_loc**: Tính toán tọa độ của ô tile dựa trên vị trí pixel của nó, bằng cách chia tọa độ pixel cho kích thước ô tile và làm tròn xuống với phần nguyên sàn (//).
- Vòng lặp: Duyệt qua tất cả các hướng xung quanh một ô tile, được định nghĩa bởi **NEIGHBOR_OFFSETS.**
- Tính toán tọa độ của ô tile xung quanh bằng cách thêm offset (độ lệch) cho tọa độ của ô hiện tại.

- Kiểm tra xem ô tile xung quanh có trong tilemap hay không bằng cách kiểm tra xem tọa độ đó có tồn tại trong **self.tilemap** hay không.
- Nếu ô tile xung quanh tồn tại trong tilemap, thêm nó vào danh sách **tiles**.

Cuối cùng, phương thức trả về danh sách tiles chứa tất cả các ô tile xung quanh vị trí cụ thể trên lưới.

```
def save(self, path):
    f = open(path, 'w')
    json.dump({'tilemap': self.tilemap, 'tile_size': self.tile_size, 'offgrid': self.offgrid_tiles}, f)
    f.close()
```

Phương thức **save** này được sử dụng để lưu tilemap vào một tệp tin JSON. Dưới đây là ý nghĩa của từng tham số và bước xử lý:

- **path**: Đường dẫn đến tệp tin JSON cần lưu.
- **f = open(path, 'w'):** Mở tệp tin với chế độ ghi.
- **json.dump(...)**: Ghi dữ liệu tilemap và các thông tin khác vào tệp tin JSON.
- **'tilemap': self.tilemap:** Lưu trữ tilemap, một từ điển ánh xạ các vị trí của các ô tile đến loại và biến thể của chúng.
- **'tile_size': self.tile**_size: Lưu trữ kích thước của mỗi ô tile.
- **'offgrid': self.offgrid_tiles**: Lưu trữ các ô tile không nằm trên lưới chính, nếu có.
- **f.close():** Đóng tệp tin sau khi đã ghi dữ liệu xong. Điều này làm cho dữ liệu được lưu trữ trong tệp tin và giải phóng tài nguyên.

```
def load(self, path):
    f = open(path, 'r')
    map_data = json.load(f)
    f.close()

    self.tilemap = map_data['tilemap']
    self.tile_size = map_data['tile_size']
    self.offgrid_tiles = map_data['offgrid']
```

Phương thức **load** này được sử dụng để tải dữ liệu tilemap từ một tệp tin JSON. Dưới đây là ý nghĩa của từng tham số và bước xử lý:

- **path**: Đường dẫn đến tệp tin JSON chứa dữ liệu tilemap cần tải.
- **f = open(path, 'r'):** Mở tệp tin ở chế độ đọc.
- **map_data = json.load(f):** Đọc dữ liệu từ tệp tin JSON và lưu trữ vào biến **map_data.**
- **f.close():** Đóng tệp tin sau khi đã đọc dữ liệu xong.
- **self.tilemap = map_data['tilemap']:** Gán dữ liệu tilemap từ **map_data** vào thuộc tính **tilemap** của đối tượng Tilemap.

- **self.tile_size = map_data['tile_size']:** Gán kích thước ô tile từ **map_data** vào thuộc tính **tile_size** của đối tượng Tilemap.
- **self.offgrid_tiles = map_data['offgrid']:** Gán danh sách các ô tile không nằm trên lưới chính từ **map_data** vào thuộc tính **offgrid_tiles** của đối tượng Tilemap.

Phương thức này cho phép đối tượng Tilemap tải dữ liệu từ một tệp tin JSON đã được lưu trước đó, giúp tái sử dụng và xử lý dữ liệu tilemap trong ứng dụng của bạn.

```
def solid_check(self, pos):
    tile_loc = str(int(pos[0] // self.tile_size)) + ';' + str(int(pos[1] // self.tile_size))
    if tile_loc in self.tilemap: #neu co ton lai location nay
        if self.tilemap[tile_loc]['type'] in PHYSICS_TILES:
            return self.tilemap[tile_loc]
```

Phương thức **solid_check** này được sử dụng để kiểm tra xem một vị trí cụ thể trên lưới có chứa một ô tile vật lý hay không. Dưới đây là ý nghĩa của từng tham số và bước xử lý:

- **pos**: Vị trí pixel cần kiểm tra.
- **tile_loc**: Tính toán tọa độ của ô tile dựa trên vị trí pixel của nó.
- **if tile_loc in self.tilemap**: Kiểm tra xem ô tile có tồn tại ở vị trí này trong tilemap không.
- Nếu ô tile tồn tại, kiểm tra xem loại của ô tile đó có trong tập hợp **PHYSICS_TILES** (các loại tile được xem là vật lý) hay không.
- Nếu có, trả về ô tile đó.

Phương thức này trả về ô tile vật lý tại vị trí cụ thể nếu nó tồn tại, hoặc None nếu không có ô tile vật lý tại vị trí đó.

```
def physics_rects_around(self, pos):
    rects = []
    for tile in self.tiles_around(pos):
        if tile['type'] in PHYSICS_TILES:
            rects.append(pygame.Rect(tile['pos'][0] * self.tile_size, tile['pos'][1] * self.tile_size, self.tile_size, self.tile_size))
    return rects
```

Phương thức **physics_rects_around** này được sử dụng để trả về danh sách các hình chữ nhật Pygame đại diện cho các ô tile vật lý xung quanh một vị trí cụ thể trên lưới. Dưới đây là ý nghĩa của từng tham số và bước xử lý:

- **pos**: Vị trí pixel của ô tile trên lưới.
- **rects**: Danh sách các hình chữ nhật Pygame đại diện cho các ô tile vật lý xung quanh vị trí cụ thể.
- Vòng lặp: Duyệt qua tất cả các ô tile xung quanh vị trí cụ thể, được trả về bởi phương thức **tiles_around.**

- Nếu loại của ô tile thuộc vào tập hợp **PHYSICS_TILES** (các loại tile được xem là vật lý), ô tile đó được thêm vào danh sách rects dưới dạng một hình chữ nhật Pygame.
- **pygame.Rect(tile['pos'][0] * self.tile_size, tile['pos'][1] * self.tile_size, self.tile_size, self.tile_size):** Tạo một hình chữ nhật Pygame từ tọa độ và kích thước của ô tile. Tọa độ và kích thước được tính dựa trên tọa độ và kích thước của ô tile trong tilemap và kích thước của mỗi ô tile.

Cuối cùng, phương thức trả về danh sách rects chứa các hình chữ nhật Pygame đại diện cho các ô tile vật lý xung quanh vị trí cụ thể trên lưới.

```
def autotile(self):
    for loc in self.tilemap:
        tile = self.tilemap[loc]
        neighbors = set()
        for shift in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            check_loc = str(tile['pos'][0] + shift[0]) + ';' + str(tile['pos'][1] + shift[1])
            if check_loc in self.tilemap:
                if self.tilemap[check_loc]['type'] == tile['type']:
                    neighbors.add(shift)
        neighbors = tuple(sorted(neighbors))
        if (tile['type'] in AUTOTILE_TYPES) and (neighbors in AUTOTILE_MAP):
            tile['variant'] = AUTOTILE_MAP[neighbors]
```

Phương thức **autotile** này được sử dụng để áp dụng kỹ thuật autotiling cho các ô tile trên lưới, cập nhật biến thể của chúng dựa trên các ô lân cận. Dưới đây là ý nghĩa của từng bước xử lý:

- Vòng lặp: Duyệt qua tất cả các ô tile trong tilemap.
- Lấy thông tin về loại và vị trí của ô tile đang xét.
- Khởi tạo một tập hợp trống **neighbors** để lưu trữ các hướng của các ô lân cận cùng loại với ô hiện tại.
- Vòng lặp **for shift in [(1, 0), (-1, 0), (0, 1), (0, -1)]::**
  - Duyệt qua tất cả các phép dịch (shift) có thể xảy ra từ ô hiện tại. Cụ thể:
    - **(1, 0):** Phép dịch sang phải.
    - **(-1, 0):** Phép dịch sang trái.
    - **(0, 1):** Phép dịch xuống dưới.
    - **(0, -1):** Phép dịch lên trên.
- Dòng mã **check_loc = str(tile['pos'][0] + shift[0]) + ';' + str(tile['pos'][1] + shift[1]):**
  - Tạo ra một chuỗi đại diện cho tọa độ của ô lân cận tương ứng với mỗi phép dịch.
  - Tọa độ này được tính bằng cách cộng tọa độ của ô hiện tại với phép dịch tương ứng. Điều này sẽ tạo ra một tọa độ mới đại diện cho ô lân cận.
    - Nếu có, thêm hướng của ô lân cận vào tập hợp neighbors.

- Dòng mã **if check_loc in self.tilemap::** Kiểm tra xem ô lân cận có tồn tại trong tilemap không. Nếu có, tiếp tục kiểm tra các điều kiện tiếp theo.
    - Dòng mã **if self.tilemap[check_loc]['type'] == tile['type']::** Kiểm tra xem loại của ô lân cận có giống loại của ô hiện tại không. Nếu có, ô này được xem là một trong các ô lân cận cùng loại và được thêm vào tập hợp **neighbors**.
- Dòng mã **neighbors = tuple(sorted(neighbors)):** Chuyển tập hợp **neighbors** thành một tuple và sắp xếp nó để nhận một key duy nhất cho từ điển **AUTOTILE_MAP.** Việc này cần thiết để đảm bảo rằng các tập hợp các hướng lân cận giống nhau sẽ có cùng một key trong **AUTOTILE_MAP.**
- Dòng mã **if (tile['type'] in AUTOTILE_TYPES) and (neighbors in AUTOTILE_MAP)::** Kiểm tra xem ô hiện tại có phải là loại tile được xác định để sử dụng kỹ thuật autotiling **(tile['type'] in AUTOTILE_TYPES)** và các ô lân cận đã được sắp xếp có tồn tại trong **AUTOTILE_MAP.** Nếu cả hai điều kiện đều đúng, tiến hành cập nhật biến thể của ô hiện tại.
- Dòng mã **tile['variant'] = AUTOTILE_MAP[neighbors]:** Gán biến thể mới cho ô tile hiện tại dựa trên giá trị từ **AUTOTILE_MAP.** Giá trị này được xác định bởi tập hợp các hướng lân cận đã sắp xếp và tìm thấy trong **AUTOTILE_MAP.**

```
def render(self, surf, offset=(0, 0)):
    for tile in self.offgrid_tiles:
        surf.blit(self.game.assets[tile['type']][tile['variant']], (tile['pos'][0] - offset[0], tile['pos'][1] - offset[1]))

    # tile['loai] la kieu tile, tile['variant'] la dang cua tile, tile['pos'] la vi tri cua tile
    # tile['variant'] dung de chon bien the cua tile do trong thu muc (vi du: grass0, grass1, grass2, ...)

    for x in range(offset[0] // self.tile_size, (offset[0] + surf.get_width()) // self.tile_size + 1):
        for y in range(offset[1] // self.tile_size, (offset[1] + surf.get_height()) // self.tile_size + 1):
            loc = str(x) + ';' + str(y)
            if loc in self.tilemap:
                tile = self.tilemap[loc]
                surf.blit(self.game.assets[tile['type']][tile['variant']], (tile['pos'][0] * self.tile_size - offset[0], tile['pos'][1] * self.tile_size - offset[1]))
```

Phương thức **render** này được sử dụng để vẽ các ô tile không nằm trên lưới chính lên bề mặt pygame. Dưới đây là ý nghĩa của từng dòng mã:

- Vòng lặp **for tile in self.offgrid_tiles::**
- Duyệt qua tất cả các ô tile không nằm trên lưới chính (**offgrid_tiles**).
- Sử dụng phương thức **blit** để vẽ ô tile lên bề mặt pygame (surf).
- **self.game.assets[tile['type']][tile['variant']]:** Truy cập đến biểu diễn hình ảnh của ô tile dựa trên loại **(tile['type'])** và biến thể **(tile['variant'])** của nó từ bộ tài nguyên trò chơi **(self.game.assets).**
- **(tile['pos'][0] - offset[0], tile['pos'][1] - offset[1]):** Xác định vị trí vẽ của ô tile trên bề mặt pygame. offset được sử dụng để điều chỉnh vị trí vẽ, đảm bảo ô tile được vẽ đúng vị trí trên bề mặt pygame.
- Vòng lặp đầu tiên **for x in range(offset[0] // self.tile_size, (offset[0] + surf.get_width()) // self.tile_size + 1)::**
    - Duyệt qua các ô tile trên trục x, bắt đầu từ ô có vị trí **offset[0] // self.tile_size** và kết thúc ở ô có vị trí **(offset[0] + surf.get_width()) // self.tile_size + 1.**

    - **offset[0] // self.tile_size**: Tính toán ô đầu tiên mà phải vẽ trên trục x, dựa trên vị trí offset và kích thước của mỗi ô tile.
    - **(offset[0] + surf.get_width()) // self.tile_size + 1**: Tính toán ô cuối cùng mà phải vẽ trên trục x, dựa trên vị trí offset, kích thước của mỗi ô tile và chiều rộng của bề mặt pygame.
    - Dòng mã này xác định một phạm vi các ô tile trên trục x mà cần được vẽ trên bề mặt pygame.
- Vòng lặp lồng nhau **for y in range(offset[1] // self.tile_size, (offset[1] + surf.get_height()) // self.tile_size + 1)::**
    - Duyệt qua các ô tile trên trục y, tương tự như trên nhưng trên trục y.
    - **loc = str(x) + ';' + str(y):** Xác định tọa độ lưới của ô tile tại vị trí hiện tại.
- Dòng mã **if loc in self.tilemap::** Kiểm tra xem ô tile tại vị trí **loc** có tồn tại trong tilemap không.
    - Nếu có, tiếp tục với các bước vẽ ô tile.
    - Nếu không, không có ô tile nào cần vẽ tại vị trí **loc**.
- Dòng mã **tile = self.tilemap[loc]:** Lấy thông tin về ô tile từ tilemap tại vị trí **loc**.
- Dòng mã **surf.blit(self.game.assets[tile['type']][tile['variant']], (tile['pos'][0] * self.tile_size - offset[0], tile['pos'][1] * self.tile_size - offset[1])):** Sử dụng phương thức **blit** để vẽ ô tile lên bề mặt pygame (**surf**).
    - **self.game.assets[tile['type']][tile['variant']]:** Truy cập đến biểu diễn hình ảnh của ô tile dựa trên loại **(tile['type'])** và biến thể **(tile['variant'])** của nó từ bộ tài nguyên trò chơi (**self.game.assets**).
    - **(tile['pos'][0] * self.tile_size - offset[0], tile['pos'][1] * self.tile_size - offset[1]):** Xác định vị trí vẽ của ô tile trên bề mặt pygame. **offset** được sử dụng để điều chỉnh vị trí vẽ, đảm bảo ô tile được vẽ đúng vị trí trên bề mặt pygame.

## 5. Entities.py (Thực thể)

Tệp entities.py có thể chứa định nghĩa và lớp cho các thực thể trong trò chơi, bao gồm nhân vật, vật phẩm, quái vật và các đối tượng tương tác khác. Các lớp trong tệp này thường đóng gói các tính năng và hành vi của các thực thể cụ thể và cung cấp phương thức để tương tác với chúng trong trò chơi.

Dưới đây là một số công dụng phổ biến mà tệp entities.py có thể thực hiện:

- **Định nghĩa các lớp thực thể**: Các lớp trong tệp này có thể định nghĩa các đặc điểm và hành vi của các thực thể khác nhau trong trò chơi, bao gồm vị trí, hình dạng, tốc độ, sức mạnh, hành vi di chuyển, và hành vi tương tác với nhân vật chính hoặc với nhau.

- **Quản lý hành vi của thực thể:** Tệp entities.py có thể chứa các phương thức để cập nhật trạng thái và hành vi của các thực thể, bao gồm cập nhật vị trí, di chuyển, xử lý va chạm và tương tác với các đối tượng khác trong trò chơi.
- **Tạo và quản lý các thực thể trong trò chơi:** Tệp entities.py cung cấp phương thức để tạo ra và quản lý các thực thể trong trò chơi, bao gồm việc tải hình ảnh, khởi tạo thực thể mới, xử lý va chạm giữa các thực thể, và loại bỏ các thực thể khi chúng không còn cần thiết.
- **Tương tác với hệ thống trò chơi khác**: Các lớp trong tệp entities.py có thể tương tác với các phần khác của hệ thống trò chơi, bao gồm hệ thống vật lý, hệ thống đồ họa và hệ thống điều khiển, để tạo ra trải nghiệm trò chơi phong phú và đa dạng.

### a. PhysicEntity (Vật lí thực thể)

```python
class PhysicsEntity:
    def __init__(self, game, e_type, pos, size):
        self.game = game
        self.type = e_type
        self.pos = list(pos)
        self.size = size
        self.velocity = [0, 0] #list chứa tốc độ ngang và dọc
        self.collision = {'up': False, 'down': False, 'right': False, 'left': False}

        self.action = ''
        self.anim_offset = (-3, -3) #kích thước nhân vật khác với hình anh animation (hit box ko dung)
        self.flip = False # nhìn sang trái hay phải
        self.set_action('idle')

        self.last_movement = [0, 0]
```

- **self.game = game:** Gán trò chơi hiện tại mà đối tượng thực thể này thuộc về.
- **self.type = e_type:** Xác định loại của đối tượng thực thể (ví dụ: nhân vật, quái vật, vật phẩm, v.v.).
- **self.pos = list(pos):** Xác định vị trí ban đầu của đối tượng thực thể trên màn hình.
- **self.size = size:** Xác định kích thước của đối tượng thực thể (chiều rộng và chiều cao).
- **self.velocity = [0, 0]:** Xác định tốc độ ban đầu của đối tượng thực thể trên cả hai trục ngang và dọc.
- **self.collision = {'up': False, 'down': False, 'right': False, 'left': False}:** Khởi tạo một từ điển để theo dõi trạng thái va chạm của đối tượng thực thể với các hướng (phía trên, dưới, trái, phải).
- **self.action = '':** Xác định hành động hiện tại của đối tượng thực thể.
- **self.anim_offset = (-3, -3):** Xác định sự chênh lệch giữa hình ảnh animation và hộp va chạm của đối tượng thực thể.
- **self.flip = False**: Xác định hướng mặt của đối tượng thực thể (hướng sang trái hoặc sang phải).

- **self.set_action('idle'):** Đặt hành động mặc định của đối tượng thực thể là "idle" (đứng yên).
- **self.last_movement = [0, 0]:** Lưu trữ tốc độ di chuyển cuối cùng của đối tượng thực thể trên cả hai trục ngang và dọc.

```
def rect(self): #dùng để phát hiện va chạm.
    return pygame.Rect(self.pos[0], self.pos[1], self.size[0], self.size[1])
```

Phương thức **rect** được sử dụng để trả về một hình chữ nhật (rectangle) Pygame, đại diện cho vùng va chạm của đối tượng thực thể.

- **return pygame.Rect(self.pos[0], self.pos[1], self.size[0], self.size[1]):** Tạo và trả về một hình chữ nhật Pygame, bắt đầu từ vị trí (pos[0], pos[1]) và có kích thước là (size[0], size[1]). Điều này cho phép sử dụng hình chữ nhật này để xác định vùng va chạm của đối tượng thực thể trong trò chơi.

```
def set_action(self, action):
    if action != self.action:
        self.action = action
        self.animation = self.game.assets[self.type + '/' + self.action].copy()
```

- Phương thức **set_action** được sử dụng để thiết lập hành động (action) mới cho đối tượng thực thể.
- **if action != self.action**: Kiểm tra xem hành động mới có khác với hành động hiện tại của đối tượng hay không. Nếu có sự thay đổi, tiếp tục thực hiện các bước sau:
- **self.action = action**: Cập nhật hành động mới cho đối tượng thực thể.
- **self.animation = self.game.assets[self.type + '/' + self.action].copy()**: Lấy animation mới từ tài nguyên trò chơi (assets) dựa trên loại và hành động mới của đối tượng thực thể. Điều này sẽ cập nhật animation cho đối tượng thực thể với hành động mới.

❖ **Kiểm tra di chuyển**

```
if movement[0] > 0:
    self.flip = False
if movement[0] < 0:
    self.flip = True
```

Dòng mã này kiểm tra hướng di chuyển của đối tượng thực thể trên trục x và cập nhật thuộc tính **flip** và **last_movement.**

- **if movement[0] > 0::** Kiểm tra xem đối tượng thực thể có di chuyển sang phải không.
- **self.flip = False**: Nếu đối tượng di chuyển sang phải, thuộc tính **flip** được thiết lập thành **False**, cho biết rằng hình ảnh của đối tượng không cần phải được lật ngược.
- **if movement[0] < 0::** Kiểm tra xem đối tượng thực thể có di chuyển sang trái không.
- **self.flip = True:** Nếu đối tượng di chuyển sang trái, thuộc tính **flip** được thiết lập thành **True**, cho biết rằng hình ảnh của đối tượng cần phải được lật ngược.
- **self.last_movement = movement:** Cập nhật self.last_movement để theo dõi hướng di chuyển của đối tượng thực thể trong frame trước đó.

❖ **Xử lí va chạm khi di chuyển**

```
def update(self, tilemap, movement = (0, 0)):
    self.collision = {'up': False, 'down': False, 'right': False, 'left': False}
    frame_movement = (movement[0] + self.velocity[0], movement[1] + self.velocity[1])
    self.pos[0] += frame_movement[0]
```

Phương thức **update** này được sử dụng để cập nhật trạng thái của đối tượng thực thể, bao gồm di chuyển, xử lý va chạm và cập nhật animation.

**self.collision = {'up': False, 'down': False, 'right': False, 'left': False}:** Khởi tạo một từ điển để theo dõi trạng thái va chạm của đối tượng thực thể với các hướng (phía trên, dưới, trái, phải).

- **frame_movement = (movement[0] + self.velocity[0], movement[1] + self.velocity[1]):** Tính toán vị trí di chuyển của đối tượng thực thể cho mỗi khung hình bằng cách kết hợp vị trí di chuyển từ bên ngoài và tốc độ di chuyển hiện tại của đối tượng.
- **self.pos[0] += frame_movement[0]:** Cập nhật vị trí của đối tượng thực thể theo trục x dựa trên vị trí di chuyển tính toán được.

```
entity_rect = self.rect()
for rect in tilemap.physics_rects_around(self.pos):
    if entity_rect.colliderect(rect):
        if frame_movement[0] > 0: #di chuyen huong qua phai
            entity_rect.right = rect.left
            self.collision['right'] = True
        if frame_movement[0] < 0:
            entity_rect.left = rect.right
            self.collision['left'] = True
        self.pos[0] = entity_rect.x
```

- **entity_rect = self.rect():** Tạo một hình chữ nhật đại diện cho vùng va chạm của đối tượng thực thể.

- **for rect in tilemap.physics_rects_around(self.pos)::** Duyệt qua tất cả các vật thể trong tilemap gần với vị trí của đối tượng thực thể.
- **if entity_rect.colliderect(rect)::** Kiểm tra xem vùng va chạm của đối tượng thực thể có va chạm với vật thể hiện tại không.
- **if frame_movement[0] > 0:** và **if frame_movement[0] < 0::** Kiểm tra hướng di chuyển của đối tượng thực thể để xác định hướng va chạm.
- **entity_rect.right = rect.left:** Cập nhật vị trí của vùng va chạm bên phải của đối tượng thực thể để nó không giao nhau với vật thể.
- **entity_rect.left = rect.right:** Cập nhật vị trí của vùng va chạm bên trái của đối tượng thực thể để nó không giao nhau với vật thể.
- **self.pos[0] = entity_rect.x:** Cập nhật lại vị trí của đối tượng thực thể dựa trên vị trí mới của vùng va chạm sau khi xử lý va chạm.

```
self.pos[1] += frame_movement[1]
entity_rect = self.rect()
for rect in tilemap.physics_rects_around(self.pos):
    if entity_rect.colliderect(rect):
        if frame_movement[1] > 0: #di chuyen huong qua phai
            entity_rect.bottom = rect.top
            self.collision['down'] = True
        if frame_movement[1] < 0:
            entity_rect.top = rect.bottom
            self.collision['up'] = True

        self.pos[1] = entity_rect.y
```

- **self.pos[1] += frame_movement[1]:** Cập nhật vị trí của đối tượng thực thể trên trục y dựa trên vị trí di chuyển tính toán được.
- **entity_rect = self.rect():** Tạo một hình chữ nhật đại diện cho vùng va chạm của đối tượng thực thể.
- **for rect in tilemap.physics_rects_around(self.pos)::** Duyệt qua tất cả các vật thể trong tilemap gần với vị trí của đối tượng thực thể.
- **if entity_rect.colliderect(rect)::** Kiểm tra xem vùng va chạm của đối tượng thực thể có va chạm với vật thể hiện tại không.
- **if frame_movement[1] > 0:** và **if frame_movement[1] < 0::** Kiểm tra hướng di chuyển của đối tượng thực thể để xác định hướng va chạm.
- **entity_rect.bottom = rect.top**: Cập nhật vị trí của vùng va chạm phía dưới của đối tượng thực thể để nó không giao nhau với vật thể.
- **entity_rect.top = rect.bottom:** Cập nhật vị trí của vùng va chạm phía trên của đối tượng thực thể để nó không giao nhau với vật thể.

- **self.pos[1] = entity_rect.y:** Cập nhật lại vị trí của đối tượng thực thể dựa trên vị trí mới của vùng va chạm sau khi xử lý va chạm.

❖ **Cập nhật thông tin về hướng di chuyển của đối tượng thực thể, tốc độ rơi, và cập nhật animation của đối tượng.**

```
self.last_movement = movement

self.velocity[1] = min(5, self.velocity[1] + 0.1) #tang toc do roi xuong

if self.collision['down'] or self.collision['up']:
    self.velocity[1] = 0


self.animation.update()
```

- **self.last_movement = movement:** Lưu trữ hướng di chuyển của đối tượng thực thể trong frame hiện tại.
- **self.velocity[1] = min(5, self.velocity[1] + 0.1):** Tăng tốc độ rơi xuống của đối tượng thực thể. Tốc độ rơi không vượt quá 5 pixel/frame.
- **if self.collision['down'] or self.collision['up']: self.velocity[1] = 0**: Kiểm tra xem đối tượng thực thể có va chạm với vật thể phía dưới hoặc phía trên không. Nếu có, đặt tốc độ dọc của đối tượng thành 0, ngăn cản việc tiếp tục rơi xuống hoặc bay lên.
- **self.animation.update():** Cập nhật animation của đối tượng thực thể để chuyển đổi giữa các khung hình.

**b. Enemy (Kẻ thù)**

```
def __init__(self, game, pos, size):
    super().__init__(game, 'enemy', pos, size)
```

Hàm **__init__()** trong lớp **Enemy** được sử dụng để khởi tạo một đối tượng kẻ địch.

- **game**: Đối tượng trò chơi, chứa tất cả thông tin và trạng thái của trò chơi.
- **pos**: Vị trí ban đầu của kẻ địch trên màn hình, được xác định bằng tọa độ (x, y).
- **size**: Kích thước của kẻ địch, thường được biểu diễn bằng chiều rộng và chiều cao của nó.

Hàm này gọi hàm khởi tạo của lớp cơ sở (**PhysicsEntity**) để thiết lập các thuộc tính cơ bản của đối tượng kẻ địch. Các tham số truyền vào là:

- **game**: Đối tượng trò chơi.

- **'enemy'**: Loại của đối tượng, trong trường hợp này là "enemy" để định danh loại của đối tượng là kẻ địch.
- **pos**: Vị trí ban đầu của đối tượng.
- **size**: Kích thước của đối tượng.

Bằng cách này, khi một đối tượng kẻ địch được tạo ra, nó sẽ có các thuộc tính và hành vi của một đối tượng vật lý trong trò chơi, cùng với các thuộc tính đặc biệt của kẻ địch.

❖ **Kiểm soát di chuyển của kẻ địch**

```
def update(self, tilemap, movement=(0, 0)):
    if self.walking:
        if tilemap.solid_check((self.rect().centerx + (-7 if self.flip else 7), self.pos[1] + 23)):
            if (self.collision['right'] or self.collision['left']):
                self.flip = not self.flip
            else:
                movement = (movement[0] - 0.5 if self.flip else 0.5, movement[1])
        else:
            self.flip = not self.flip
        self.walking = max(0, self.walking - 1)
```

- Dòng **if self.walking:** kiểm tra xem kẻ địch có đang trong trạng thái di chuyển không. Nếu có, nó sẽ thực hiện các thao tác kiểm tra va chạm và điều chỉnh hướng di chuyển.
- Dòng **if tilemap.solid_check((self.rect().centerx + (-7 if self.flip else 7), self.pos[1] + 23))**: kiểm tra xem có va chạm giữa kẻ địch và bản đồ không, dựa trên điểm kiểm tra được tính toán trước đó. Nếu có va chạm, nó sẽ xác định hướng di chuyển tiếp theo của kẻ địch.
    - **self.rect().centerx**: Là tọa độ trung tâm theo trục X của hộp va chạm của kẻ địch.
    - **self.flip:** Biến boolean xác định hướng của kẻ địch, có giá trị True nếu đang nhìn về phía trái và False nếu đang nhìn về phía phải.
    - **self.pos[1] + 23**: Tọa độ Y của kẻ địch cộng với một giá trị cố định (23 pixel). Điểm này có thể nằm ở phía dưới hoặc trên của hộp va chạm, tùy thuộc vào cách xử lý va chạm cụ thể trong game.
- Nếu không có va chạm, kẻ địch sẽ đảo hướng đi lại. Điều này giúp kẻ địch tránh va chạm với các vật cản.
- Cuối cùng, biến **self.walking** sẽ được giảm đi 1 đơn vị, biểu thị thời gian di chuyển còn lại của kẻ địch. Nếu **self.walking** đạt giá trị 0 hoặc âm, kẻ địch sẽ dừng lại và không thực hiện di chuyển tiếp theo.

```
    if not self.walking:
        dis = (self.game.player.pos[0] - self.pos[0], self.game.player.pos[1] - self.pos[1])
        if (abs(dis[1]) < 16):
            if (self.flip and dis[0] < 0):
                self.game.projectiles.append([[self.rect().centerx - 7, self.rect().centery], -1.5, 0])
                for i in range(4):
                    self.game.sparks.append(Spark(self.game.projectiles[-1][0], random.random() - 0.5 + math.pi, 2 + random.random()))
            if (not self.flip and dis[0] > 0):
                self.game.projectiles.append([[self.rect().centerx + 7, self.rect().centery], 1.5, 0])
                for i in range(4):
                    self.game.sparks.append(Spark(self.game.projectiles[-1][0], random.random() - 0.5, 2 + random.random()))
elif random.random() < 0.01:
    self.walking = random.randint(30, 120)
```

Nếu kẻ địch không đang di chuyển (**not self.walking**) và người chơi nằm trong phạm vi gần của nó (nằm trong hình vuông với cạnh là 16 pixel tính từ vị trí của kẻ địch).

Nếu điều kiện này đúng, kẻ địch sẽ bắn các viên đạn theo hướng của người chơi. Nếu kẻ địch đang hướng về bên trái (**self.flip**) và người chơi đang ở bên trái của nó, nó sẽ bắn ra bên trái với tốc độ -1.5. Ngược lại, nếu kẻ địch đang hướng về bên phải và người chơi đang ở bên phải của nó, nó sẽ bắn ra bên phải với tốc độ 1.5.

Ngoài ra, nó cũng tạo ra một số tia lửa (**Spark**) để tạo hiệu ứng hình ảnh.

Nếu điều kiện trên không đúng, có một xác suất rất nhỏ (xác suất là 0.01) rằng kẻ địch sẽ bắt đầu di chuyển một lần nữa, với thời gian di chuyển được chọn ngẫu nhiên trong khoảng từ 30 đến 120 frames.

- **if not self.walking:**
  - Kiểm tra xem kẻ địch có đang di chuyển không.
- **dis = (self.game.player.pos[0] - self.pos[0], self.game.player.pos[1] - self.pos[1]):**
  - Tính toán khoảng cách giữa vị trí của kẻ địch và vị trí của người chơi.
- **if (abs(dis[1]) < 16):**
  - Kiểm tra xem khoảng cách theo trục y giữa kẻ địch và người chơi có nhỏ hơn 16 pixel hay không. Nếu nhỏ hơn, có nghĩa là người chơi nằm trong phạm vi tấn công của kẻ địch theo chiều dọc.
- **if (self.flip and dis[0] < 0):**
  - Kiểm tra xem kẻ địch có đang hướng về phía bên trái và người chơi có đang ở bên trái của nó không.
- **self.game.projectiles.append([[self.rect().centerx - 7, self.rect().centery], -1.5, 0])**
  - Thêm một viên đạn vào danh sách đạn trong trò chơi. Đạn này sẽ được bắn từ vị trí trung tâm của kẻ địch, với tốc độ di chuyển theo hướng âm x với tốc độ -1.5.
- **if (not self.flip and dis[0] > 0):**
  - Kiểm tra xem kẻ địch có đang hướng về phía bên phải và người chơi có đang ở bên phải của nó không.
- **self.game.projectiles.append([[self.rect().centerx + 7, self.rect().centery], 1.5, 0])**
  - Thêm một viên đạn vào danh sách đạn trong trò chơi. Đạn này sẽ được bắn từ vị trí trung tâm của kẻ địch, với tốc độ di chuyển theo hướng dương x với tốc độ 1.5.

- **elif random.random() < 0.01:**
  - Nếu không trong trạng thái di chuyển, có một xác suất rất nhỏ (xác suất là 0.01) rằng kẻ địch sẽ bắt đầu di chuyển lại.
- **self.walking = random.randint(30, 120)**
  - Nếu kẻ địch quyết định bắt đầu di chuyển lại, thời gian di chuyển sẽ được chọn ngẫu nhiên trong khoảng từ 30 đến 120 frames.

❖ **Tạo tương tác kẻ địch bị tấn công bởi người chơi**

```
super().update(tilemap, movement=movement)

if movement[0] != 0:
    self.set_action('run')
else:
    self.set_action('idle')

if abs(self.game.player.dashing) >= 50: #dashing
    if self.rect().colliderect(self.game.player.rect()): #hit some thing
        for i in range(30):
            angle = random.random() * math.pi * 2
            speed = random.random() * 5
            self.game.sparks.append(Spark(self.rect().center, angle, 2 + random.random()))
            self.game.particles.append(Particle(self.game, 'particle', self.rect().center, velocity=[math.cos(angle + math.pi) *
                speed * 0.5, math.sin(angle + math.pi) * speed * 0.5], frame=random.randint(0, 7)))
        self.game.sparks.append(Spark(self.rect().center, 0, 5 + random.random()))
        self.game.sparks.append(Spark(self.rect().center, math.pi, 5 + random.random()))
        return True
```

Dòng code này gọi phương thức **update** của lớp cơ sở **PhysicsEntity** và truyền vào tham số **tilemap** và **movement=movement.**

Nếu giá trị của **movement[0]** không bằng 0, tức là kẻ địch đang di chuyển ngang, thì nó đặt hành động của kẻ địch thành "run", ngược lại nó đặt hành động thành "idle".

- Nếu giá trị tuyệt đối của **self.game.player.dashing** lớn hơn hoặc bằng 50 (nghĩa là người chơi đang thực hiện một cú dash), và hình chữ nhật của kẻ địch va chạm với hình chữ nhật của người chơi, thì nó tạo ra một loạt hiệu ứng hạt và tia lửa xung quanh vị trí của kẻ địch và trả về True để thông báo rằng kẻ địch đã bị tấn công.

### c. Player (Người chơi)

```
def __init__(self, game, pos, size):
    super().__init__(game, 'player', pos, size)
    self.air_time = 0 #thoi gian dang o tren khong
    self.jumps = 1 #co the nhay hay khong/so lan nhay (neu cham tuong dc reset lan nhay)
    self.wall_slide = False
    self.dashing = 0
```

Phương thức **__init__** này khởi tạo một đối tượng Player với các thuộc tính cơ bản như vị trí, kích thước và loại.

- **game**: Tham chiếu đến trò chơi hiện tại.

- **pos**: Vị trí ban đầu của player trên màn hình.
- **size**: Kích thước của player.
- **air_time**: Thời gian player đã ở trên không.
- **jumps**: Số lần nhảy có sẵn cho player.
- **wall_slide**: Biến xác định liệu player có thể trượt trên tường không.
- **dashing**: Trạng thái của hành động dash của player. Giá trị này được sử dụng để quản lý thời gian và hướng dash.

❖ **Hiệu ứng leo tường của player**

```
def __init__(self, game, pos, size):
    super().__init__(game, 'player', pos, size)
    self.air_time = 0 #thoi gian dang o tren khong
    self.jumps = 1 #co the nhay hay khong/so lan nhay (neu cham tuong dc reset lan nhay)
    self.wall_slide = False
    self.dashing = 0
```

Sau khi cập nhật trạng thái của player dựa trên tilemap và movement, đoạn code này tiếp tục kiểm tra xem player có đang tiếp xúc với tường ở phía trái hoặc phải không và có đang trong trạng thái nằm trên không ít nhất 4 frame không. Nếu cả hai điều kiện đều đúng, player sẽ bắt đầu trượt trên tường.

Nếu player đang tiếp xúc với tường ở phía bên phải, flip của player sẽ được thiết lập là False, ngược lại, nếu player đang tiếp xúc với tường ở phía bên trái, flip sẽ được thiết lập là True.

Dòng đầu tiên gọi phương thức update của lớp cha với các tham số **tilemap** và **movement=movement**. Dòng thứ hai tăng biến **air_time** lên 1 đơn vị.

Dòng thứ ba kiểm tra xem player có đang tiếp xúc với mặt đất không **(self.collision['down']).** Nếu đang, **air_time** được đặt về 0 và số lượng lần nhảy lại được đặt về 1.

Dòng thứ tư thiết lập **wall_slide** về False, chỉ định rằng player không đang trượt trên tường.

Dòng thứ năm kiểm tra xem player có đang tiếp xúc với tường ở bên phải hoặc bên trái không và đã trôi qua ít nhất 4 frame kể từ lần cuối cùng player chạm mặt đất (**self.air_time > 4**).

Nếu cả hai điều kiện đều đúng, **self.wall_slide** được thiết lập thành True, chỉ định rằng player đang trượt trên tường. Đồng thời, **self.velocity[1]** được thiết lập không vượt quá 0.5, giúp player giảm tốc độ khi đang trượt.

Nếu player đang tiếp xúc với tường bên phải (**self.collision['right']**), **self.flip** được thiết lập thành False, chỉ định rằng player không quay đầu về phía bên trái

Dòng cuối cùng thiết lập hành động của player thành **'wall_slide'**.

❖ **Kết thúc hiệu ứng leo tường của player**

```
if not self.wall_slide:
    if self.air_time > 4:
        self.set_action('jump')
    elif movement[0] != 0:
        self.set_action('run')
    else:
        self.set_action('idle')
```

Nếu player không đang trượt trên tường (**not self.wall_slide**), kiểm tra các trường hợp sau:

Nếu **self.air_time** (thời gian ở trên không) lớn hơn 4, player được thiết lập thành hành động "jump" (nhảy).

Nếu di chuyển ngang của player khác không (**movement[0]** không bằng 0), player được thiết lập thành hành động "run" (chạy).

Trong trường hợp còn lại, player được thiết lập thành hành động "**idle**" (đứng yên).

❖ **Chức năng Dashing (Lướt) của player.**

```
if abs(self.dashing) in {60, 50}: #bat dau hay ket thuc dash
    for i in range(20):
        angle = random.random() * math.pi * 2
        speed = random.random() * 0.5 + 0.5 #chon 1 so giua 0.5 va 1
        pvelocity = [math.cos(angle) * speed, math.sin(angle) * speed]
        self.game.particles.append(Particle(self.game, 'particle', self.rect().center, velocity=pvelocity, frame=random.randint(0, 7)))
if self.dashing > 0:
    self.dashing = max(0, self.dashing - 1)
if self.dashing < 0:
    self.dashing = min(0, self.dashing + 1)
if abs(self.dashing) > 50:
    self.velocity[0] = abs(self.dashing) / self.dashing * 8 #frame dau cua dash (co doan code la phan cooldown cua dash nua ?????)
    if abs(self.dashing) == 51: # ket thuc dash
        self.velocity[0] *= 0.1
    pvelocity = [abs(self.dashing) / self.dashing * random.random() * 3, 0] #random num from 0 to 3, k dong den y
    self.game.particles.append(Particle(self.game, 'particle', self.rect().center, velocity=pvelocity, frame=random.randint(0, 7)))
```

Dòng đầu tiên kiểm tra xem giá trị tuyệt đối của biến **self.dashing** có trong tập hợp **{60, 50}** hay không. Điều này ám chỉ rằng người chơi đang bắt đầu hoặc kết thúc việc dash.

Nếu điều kiện đúng, vòng lặp được thực thi 20 lần. Trong mỗi lần lặp, một góc và tốc độ được chọn ngẫu nhiên, sau đó được sử dụng để tạo ra một vector vận tốc mới

**pvelocity** trên mặt phẳng 2D. Vector này được sử dụng để tạo ra các hạt (particles) cho hiệu ứng hạt phát ra từ người chơi khi họ thực hiện dash.

Sau đó, ba điều kiện tiếp theo kiểm tra giá trị của biến **self.dashing**. Nếu **self.dashing** lớn hơn 0, nó được giảm đi 1 đơn vị (để tính thời gian còn lại của dash). Nếu **self.dashing** nhỏ hơn 0, nó được tăng lên 1 đơn vị (để tính thời gian còn lại của dash).

Nếu giá trị tuyệt đối của **self.dashing** lớn hơn 50, người chơi vẫn đang trong trạng thái dash. Trong trường hợp này, tốc độ ngang của người chơi được đặt bằng giá trị tuyệt đối của **self.dashing** chia cho chính nó nhân với 8. Điều này giúp người chơi di chuyển với tốc độ nhanh nhất khi đang dash. Nếu giá trị tuyệt đối của **self.dashing** là 51 (điều kiện này xảy ra khi dash kết thúc), tốc độ ngang của người chơi được giảm đi một phần mười. Cuối cùng, một vector vận tốc mới được tạo ra ngẫu nhiên trong một phạm vi cụ thể và được sử dụng để tạo ra các hạt khác cho hiệu ứng.

```
if self.velocity[0] > 0:
    self.velocity[0] = max(0, self.velocity[0] - 0.1)
else:
    self.velocity[0] = min(0, self.velocity[0] + 0.1)
```

Dòng code này điều chỉnh tốc độ ngang của đối tượng người chơi. Nó kiểm tra xem tốc độ ngang có lớn hơn 0 không. Nếu đúng, nó giảm tốc độ ngang đi 0.1 đơn vị, nhưng tối thiểu là 0 (không âm). Nếu tốc độ ngang nhỏ hơn hoặc bằng 0, nó tăng tốc độ ngang lên 0.1 đơn vị, nhưng tối đa là 0 (không dương). Điều này đảm bảo rằng tốc độ ngang luôn được giữ trong một khoảng giá trị nhất định, không cho phép nó phát triển quá lớn theo cả hai hướng.

```
def render(self, surf, offset=(0, 0)):
    if abs(self.dashing) <= 50: # khuc nay la dang cooldown cai dash hoac la ko muon dash
        super().render(surf, offset=offset) # cu render nhu cu thoi :))
```

Dòng này là phần render của đối tượng người chơi. Nó kiểm tra xem trạng thái của việc dash có đang trong giai đoạn cooldown (hoặc người chơi không muốn dash) hay không. Nếu giá trị tuyệt đối của **self.dashing** nhỏ hơn hoặc bằng 50, điều này ngụ ý rằng đối tượng đang trong giai đoạn cooldown của dash hoặc không muốn thực hiện dash. Trong trường hợp này, nó gọi phương thức render của lớp cha (**PhysicsEntity**) để vẽ hình ảnh của đối tượng người chơi lên màn hình.

```
def dash(self):
    if not self.dashing:
        if self.flip:
            self.dashing = -60
        else:
            self.dashing = 60
```

Hàm **dash** được gọi khi người chơi thực hiện hành động **dash**. Nó kiểm tra xem có đang trong trạng thái dash hay không. Nếu không, nó xác định hướng của nhân vật (nếu nhân vật đang hướng về bên phải, dashing sẽ có giá trị là -60, ngược lại, nếu nhân vật hướng về bên trái, dashing sẽ có giá trị là 60). Điều này sẽ tạo ra hiệu ứng dash với vận tốc nhanh hơn về hướng tương ứng.

❖ **Chức năng Jump (Nhảy) của player.**

```
def jump(self):
    if self.wall_slide:
        if self.flip and self.last_movement[0] < 0:
            self.velocity[0] = 3.5 #thay 1.5 di toi cua tuong lai no on hay cai cu lin 3.5 nay
            self.velocity[1] = -2.5
            self.air_time = 5
            self.jumps = max(self.jumps - 1, 0)
            return True
        elif not self.flip and self.last_movement[0] > 0:
            self.velocity[0] = -3.5
            self.velocity[1] = -2.5
            self.air_time = 5
            self.jumps = max(self.jumps - 1, 0)
            return True

    elif self.jumps:
        self.velocity[1] = -3
        self.jumps -= 1
        self.air_time = 5
        return True
```

- Nếu đối tượng đang trượt tường (**wall_slide**), nó kiểm tra hướng của lần di chuyển trước đó của đối tượng (**last_movement**). Nếu đối tượng đã hướng về phía trái và di chuyển từ trái sang phải (**flip** là **True** và **last_movement[0]** là âm), nó sẽ đặt vận tốc theo phương x là 3.5 và phương y là -2.5 để nhảy ra xa tường. Ngược lại, nếu đối tượng đã hướng về phía phải và di chuyển từ phải sang trái (flip là False và **last_movement[0]** là dương), nó sẽ đặt vận tốc theo phương x là -3.5 và phương y là -2.5 để nhảy ra xa tường. Sau đó, nó giảm số lần nhảy còn lại và trả về True.
- Nếu không, nếu đối tượng còn số lần nhảy (**jumps** khác 0), nó đặt vận tốc theo phương y là -3 để thực hiện một nhảy thường. Sau đó, nó giảm số lần nhảy còn lại và trả về True.

## 6. Utils.py (Những tiện ích trong quá trình làm code)

Đây là một hàm khá là quan trọng trong qúa trình xây đừng đồ án game. Nó bao gồm việc chuyển đổi nhận diện hình ảnh và xóa phông cho hình ảnh để tiện add vào game play, Nó còn xử lý số ảnh được chạy trong 1 giây:

```
def load_image(path):
    img = pygame.image.load(BASE_IMG_PATH + path).convert() #hieu nang tro choi
    #Chuyển đổi định dạng: Gọi .convert() để chuyển đổi hình ảnh sang định dạng p
    img.set_colorkey((255, 255, 255))
    return img
```

**img = pygame.image.load(BASE_IMG_PATH + path).convert():** Tải hình ảnh từ đường dẫn **BASE_IMG_PATH + path** và chuyển đổi nó thành định dạng pixel phù hợp với Pygame sử dụng **.convert().** Điều này giúp tăng hiệu năng hiển thị của hình ảnh.

- **img.set_colorkey((255, 255, 255)):** Thiết lập màu trắng **(255, 255, 255)** là màu chính trong hình ảnh sẽ trở thành màu trong suốt, cho phép vẽ hình ảnh mà không cần vẽ màu nền trắng. Điều này thường được sử dụng khi bạn muốn vẽ hình ảnh với các đường viền không đều.
- Cuối cùng, hàm trả về hình ảnh đã được tải và xử lý.

```
def load_images(path):
    images = []
    for img_name in sorted(os.listdir(BASE_IMG_PATH + path)): #lay tat ca ten file trong path
        images.append(load_image(path + '/' + img_name))

    return images
```

Hàm **load_images** này được sử dụng để tải nhiều hình ảnh từ một thư mục cụ thể. Dưới đây là ý nghĩa của từng dòng mã:

- **images = []:** Khởi tạo một danh sách để lưu trữ các hình ảnh.
- **for img_name in sorted(os.listdir(BASE_IMG_PATH + path)):**: Duyệt qua tất cả các tệp hình ảnh trong thư mục **BASE_IMG_PATH + path. os.listdir()** trả về một danh sách các tên tệp trong thư mục được chỉ định **.sorted()** được sử dụng để đảm bảo rằng các hình ảnh được tải theo thứ tự được sắp xếp.
- **images.append(load_image(path + '/' + img_name)):** Thêm hình ảnh được tải từ đường dẫn **path + '/' + img_name** vào danh sách **images** bằng cách sử dụng hàm **load_image** đã được định nghĩa trước đó.
- Cuối cùng, hàm trả về danh sách **images** chứa tất cả các hình ảnh đã được tải và xử lý từ thư mục cụ thể.

```python
class Animation:
    def __init__(self, images, img_dur = 5, loop = True):
        self.images = images
        self.loop = loop
        self.img_duration = img_dur # mỗi hình ảnh được hiển thị bao lâu (5 khung hình cho 1 ảnh mặc định)
        self.done = False
        self.frame = 0

    def copy(self):
        return Animation(self.images, self.img_duration, self.loop)
```

Trong hàm **__init__,** một đối tượng Animation được khởi tạo với các thuộc tính sau:

- **images**: Một danh sách các hình ảnh đại diện cho các khung hình của animation.
- **img_dur**: Thời lượng hiển thị mỗi hình ảnh trong animation, mặc định là 5 frames.
- **loop**: Biến boolean xác định xem animation có lặp lại hay không, mặc định là True.
- **done**: Biến boolean xác định xem animation đã hoàn thành hay không, mặc định là False.
- **frame**: Chỉ số của khung hình hiện tại trong danh sách **images**, mặc định là 0.

Phương thức **copy** được sử dụng để tạo một bản sao của đối tượng Animation với cùng các thuộc tính và giá trị. Bản sao này có thể được sử dụng để tạo ra một animation tương tự nhưng độc lập với đối tượng gốc.

```python
def update(self):
    if self.loop:
        self.frame = (self.frame + 1) % (self.img_duration * len(self.images))
    else:
        self.frame = min(self.frame + 1, self.img_duration * len(self.images) - 1)
        if self.frame >= self.img_duration * len(self.images) - 1:
            self.done = True
```

- Nếu **self.loop** là True, nghĩa là animation sẽ lặp lại:
- **self.frame = (self.frame + 1) % (self.img_duration * len(self.images)):** Tăng chỉ số của frame hiện tại lên một đơn vị, sau đó sử dụng toán tử % để đảm bảo rằng chỉ số frame luôn nằm trong phạm vi từ 0 đến **(self.img_duration * len(self.images)) - 1**. Điều này đảm bảo rằng animation sẽ lặp lại khi chỉ số frame vượt quá số lượng frames của animation.
- Nếu **self.loop** là False, nghĩa là animation không lặp lại:
- **self.frame = min(self.frame + 1, self.img_duration * len(self.images) - 1):** Tăng chỉ số của frame hiện tại lên một đơn vị, nhưng không vượt quá số lượng frames cuối cùng của animation.
- **if self.frame >= self.img_duration * len(self.images) - 1: self.done = True:** Nếu chỉ số frame đã đạt đến hoặc vượt quá số lượng frames cuối cùng của animation,

đánh dấu animation đã hoàn thành bằng cách đặt self.done thành True. Điều này chỉ xảy ra khi animation không lặp lại và đã chạy đến hết.

```
def img(self):
    return self.images[int(self.frame / self.img_duration)]
```

- **return self.images[int(self.frame / self.img_duration)]:** Tính chỉ số của frame hiện tại trong danh sách hình ảnh (**self.images**) bằng cách chia **self.frame** cho **self.img_duration** và lấy phần nguyên của kết quả. Điều này sẽ cho chúng ta biết frame hiện tại là frame nào trong danh sách hình ảnh. Sau đó, trả về hình ảnh tương ứng với frame đó.

## 7. Particle.py (Hiệu ứng hạt xung quanh nhân vật)

Là hiệu ứng xung quanh nhân vật khi sử dụng chiêu thức bao gồm những các hạt vây quanh

Phương thức **__init__** này được sử dụng để khởi tạo một đối tượng hạt phát ra từ game.

- **self.game = game:** Lưu trữ tham chiếu đến đối tượng game mà hạt này thuộc về.
- **self.type = p_type:** Xác định loại hạt bằng cách sử dụng chuỗi **p_type**.
- **self.pos = list(pos):** Lưu trữ vị trí ban đầu của hạt. Biến pos có thể là một bộ giá trị (x, y), vì vậy nó được chuyển đổi thành một danh sách để dễ dàng sửa đổi.
- **self.velocity = list(velocity**): Lưu trữ vận tốc ban đầu của hạt. Biến **velocity** cũng có thể là một bộ giá trị (vx, vy), vì vậy nó được chuyển đổi thành một danh sách để dễ dàng sửa đổi.
- **self.animation = self.game.assets['particle/' + p_type].copy():** Tạo một bản sao của animation tương ứng với loại hạt từ tệp tài nguyên của game.
- **self.animation.frame = frame:** Thiết lập frame hiện tại của animation thành giá trị **frame**, cho phép chọn frame khởi đầu của animation nếu cần.

```
    def update(self):
        kill = False
        if self.animation.done:
            kill = True
        self.pos[0] += self.velocity[0]
        self.pos[1] += self.velocity[1]

        self.animation.update()

        return kill
```

- kill = False: Khởi tạo biến kill thành False. Biến này sẽ chỉ định xem hạt có nên bị loại bỏ khỏi trò chơi không.
- if self.animation.done: kill = True: Kiểm tra xem animation của hạt đã hoàn thành chưa. Nếu đã hoàn thành (done = True), gán kill thành True để đánh dấu rằng hạt cần bị loại bỏ khỏi trò chơi.
- self.pos[0] += self.velocity[0] và self.pos[1] += self.velocity[1]: Cập nhật vị trí của hạt dựa trên vận tốc hiện tại của nó.
- self.animation.update(): Cập nhật animation của hạt.
- return kill: Trả về giá trị của biến kill, cho biết liệu hạt cần bị loại bỏ khỏi trò chơi hay không.

```
def render(self, surf, offset=(0, 0)):
    img = self.animation.img()
    surf.blit(img, (self.pos[0] - offset[0] - img.get_width() // 2, self.pos[1] - offset[1] - img.get_height() // 2))
```

- Phương thức **render** này được sử dụng để vẽ hạt lên bề mặt pygame.
- **img = self.animation.img():** Lấy hình ảnh hiện tại của animation của hạt.
- **surf.blit(img, (self.pos[0] - offset[0] - img.get_width() // 2, self.pos[1] - offset[1] - img.get_height() // 2)):** Vẽ hình ảnh của hạt lên bề mặt pygame.
- **self.pos[0] - offset[0] - img.get_width() // 2**: Tọa độ x của hạt được tính toán bằng cách trừ đi độ lệch tọa độ x (offset), sau đó trừ đi nửa chiều rộng của hình ảnh để căn giữa hạt.
- **self.pos[1] - offset[1] - img.get_height() // 2:** Tương tự, tọa độ y của hạt được tính toán bằng cách trừ đi độ lệch tọa độ y (offset), sau đó trừ đi nửa chiều cao của hình ảnh để căn giữa hạt.
- Cuối cùng, hình ảnh của hạt được vẽ lên bề mặt pygame.

## 8. spark.py

Dùng để tạo tia lửa cho Quái vật.

```
def __init__(self, pos, angle, speed):
    self.pos = list(pos)
    self.angle = angle
    self.speed = speed
```

Phương thức **__init__** này được sử dụng để khởi tạo một đối tượng vận tốc với các thuộc tính cơ bản. Dưới đây là ý nghĩa của từng tham số:

- **pos**: Vị trí ban đầu của đối tượng, được biểu diễn dưới dạng một cặp số tọa độ (x, y).
- **angle**: Hướng di chuyển ban đầu của đối tượng, được đo bằng góc với trục x dương, được tính bằng đơn vị độ.
- **speed**: Tốc độ di chuyển ban đầu của đối tượng, được biểu diễn bằng một số không âm.

Các giá trị này được sử dụng để tạo ra một đối tượng vận tốc với vị trí ban đầu, hướng di chuyển và tốc độ ban đầu đã xác định.

```
def __init__(self, pos, angle, speed):
    self.pos = list(pos)
    self.angle = angle
    self.speed = speed
```

Phương thức **update** này cập nhật vị trí của đối tượng dựa trên hướng và tốc độ hiện tại, sau đó giảm tốc độ theo thời gian. Dưới đây là ý nghĩa của từng dòng mã:

- Dòng **self.pos[0] += math.cos(self.angle) * self.speed**:
- Cập nhật vị trí của đối tượng theo hướng ngang (trục x) bằng cách thêm giá trị delta x, được tính bằng cosin của góc di chuyển nhân với tốc độ.
- Dòng **self.pos[1] += math.sin(self.angle) * self.speed:**
- Cập nhật vị trí của đối tượng theo hướng dọc (trục y) tương tự như trên, nhưng thay vì sử dụng cosin, ta sử dụng sin.

- Dòng **self.speed = max(0, self.speed - 0.1):**
- Giảm tốc độ của đối tượng đi một lượng nhất định sau mỗi lần cập nhật. Giá trị tốc độ sẽ không bao giờ nhỏ hơn 0.
- Dòng **return not self.speed:**
- Trả về True nếu tốc độ của đối tượng đã giảm xuống 0, ngược lại trả về False. Điều này có thể được sử dụng để kiểm tra xem đối tượng đã dừng lại hay chưa.

```
def render(self, surf, offset=(0,0)):
    render_points = [
        (self.pos[0] + math.cos(self.angle) * self.speed * 3 - offset[0], self.pos[1] + math.sin(self.angle) * self.speed * 3 - offset[1]),
        (self.pos[0] + math.cos(self.angle + math.pi * 0.5) * self.speed * 0.5 - offset[0], self.pos[1] + math.sin(self.angle + math.pi * 0.5) * self.speed * 0.5 - offset[1]),
        (self.pos[0] + math.cos(self.angle + math.pi) * self.speed * 3 - offset[0], self.pos[1] + math.sin(self.angle + math.pi) * self.speed * 3 - offset[1]),
        (self.pos[0] + math.cos(self.angle - math.pi * 0.5) * self.speed * 0.5 - offset[0], self.pos[1] + math.sin(self.angle - math.pi * 0.5) * self.speed * 0.5 - offset[1]),
    ]

    pygame.draw.polygon(surf, (255, 255, 255), render_points)
```

Phương thức **render** này được sử dụng để vẽ đối tượng di chuyển trên bề mặt pygame (surf). Dưới đây là ý nghĩa của từng dòng mã:

- Dòng mã **render_points = [...]**: Xác định các điểm cần vẽ để tạo thành hình ảnh của đối tượng di chuyển.
- Mỗi điểm được tính toán bằng cách sử dụng công thức hình polar (hay còn gọi là hình cực) để tính toán vị trí mới của các điểm dựa trên vị trí, hướng, và tốc độ của đối tượng.
- **math.cos** và **math.sin** được sử dụng để tính toán tọa độ của các điểm dựa trên góc và tốc độ của đối tượng.
- **math.pi** được sử dụng để xác định góc quay 90 độ và 180 độ.
- Dòng mã **pygame.draw.polygon(surf, (255, 255, 255), render_points):** Vẽ một đa giác với các điểm đã tính toán lên bề mặt pygame (surf).
- Màu trắng (**255, 255, 255**) được sử dụng cho các đường vẽ.
- Các điểm **render_points** được sử dụng để xác định hình dạng của đa giác.

- **(self.pos[0] + math.cos(self.angle) * self.speed * 3 - offset[0], self.pos[1] + math.sin(self.angle) * self.speed * 3 - offset[1]):**
  - **self.pos[0] + math.cos(self.angle) * self.speed * 3:** Tọa độ x mới của điểm này được tính toán bằng cách cộng tọa độ x hiện tại của đối tượng với phần delta x được tính bằng cách nhân cosin của góc di chuyển với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 3).
  - **self.pos[1] + math.sin(self.angle) * self.speed * 3**: Tương tự, tọa độ y mới của điểm này được tính toán bằng cách cộng tọa độ y hiện tại của đối tượng với phần delta y được tính bằng cách nhân sin của góc di chuyển với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 3).
  - **offset[0]:** Độ lệch tọa độ x, thường được sử dụng để điều chỉnh vị trí của toàn bộ bề mặt vẽ.
  - **offset[1]:** Độ lệch tọa độ y, tương tự như trên.

Tổng cùng với **offset** sẽ đưa ra tọa độ cuối cùng của điểm này trên bề mặt pygame, được sử dụng để vẽ.

- **(self.pos[0] + math.cos(self.angle + math.pi * 0.5) * self.speed * 0.5 - offset[0], self.pos[1] + math.sin(self.angle + math.pi * 0.5) * self.speed * 0.5 - offset[1]):**
  - **self.pos[0] + math.cos(self.angle + math.pi * 0.5) * self.speed * 0.5**: Tọa độ x mới của điểm này được tính bằng cách cộng tọa độ x hiện tại của đối tượng với phần delta x được tính bằng cách nhân cosin của góc di chuyển (+90 độ) với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 0.5).
  - **self.pos[1] + math.sin(self.angle + math.pi * 0.5) * self.speed * 0.5**: Tương tự, tọa độ y mới của điểm này được tính bằng cách cộng tọa độ y hiện tại của đối tượng với phần delta y được tính bằng cách nhân sin của góc di chuyển (+90 độ) với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 0.5).
  - **offset[0]:** Độ lệch tọa độ x, thường được sử dụng để điều chỉnh vị trí của toàn bộ bề mặt vẽ.
  - **offset[1]:** Độ lệch tọa độ y, tương tự như trên.

- **(self.pos[0] + math.cos(self.angle + math.pi) * self.speed * 3 - offset[0], self.pos[1] + math.sin(self.angle + math.pi) * self.speed * 3 - offset[1]):**
  - **self.pos[0] + math.cos(self.angle + math.pi) * self.speed * 3:** Tọa độ x mới của điểm này được tính bằng cách cộng tọa độ x hiện tại của đối tượng với phần delta x được tính bằng cách nhân cosin của góc di chuyển (+180 độ) với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 3).
  - **self.pos[1] + math.sin(self.angle + math.pi) * self.speed * 3:** Tương tự, tọa độ y mới của điểm này được tính bằng cách cộng tọa độ y hiện tại của đối tượng với phần delta y được tính bằng cách nhân sin của góc di chuyển (+180 độ) với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 3).
  - **offset[0]:** Độ lệch tọa độ x, thường được sử dụng để điều chỉnh vị trí của toàn bộ bề mặt vẽ.
  - **offset[1]:** Độ lệch tọa độ y, tương tự như trên.

- **(self.pos[0] + math.cos(self.angle - math.pi * 0.5) * self.speed * 0.5 - offset[0], self.pos[1] + math.sin(self.angle - math.pi * 0.5) * self.speed * 0.5 - offset[1]):**
  - **self.pos[0] + math.cos(self.angle - math.pi * 0.5) * self.speed * 0.5**: Tọa độ x mới của điểm này được tính bằng cách cộng tọa độ x hiện tại của đối tượng với phần delta x được tính bằng cách nhân cosin của góc di chuyển (-90 độ) với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 0.5).

- **self.pos[1] + math.sin(self.angle - math.pi * 0.5) * self.speed * 0.5:** Tương tự, tọa độ y mới của điểm này được tính bằng cách cộng tọa độ y hiện tại của đối tượng với phần delta y được tính bằng cách nhân sin của góc di chuyển (-90 độ) với tốc độ, sau đó nhân với một hệ số tỉ lệ (ở đây là 0.5).
- **offset[0]:** Độ lệch tọa độ x, thường được sử dụng để điều chỉnh vị trí của toàn bộ bề mặt vẽ.
- **offset[1]:** Độ lệch tọa độ y, tương tự như trên.

## 9. Editor.py

Khởi tạo một số thuộc tính cho một trò chơi hoặc ứng dụng đồ họa.

```
self.movement = [False, False, False, False]

self.tilemap = Tilemap(self, tile_size= 16)


try:
    self.tilemap.load('map.json')
except FileNotFoundError:
    pass
self.scroll = [0, 0] #camera pos in top left

self.tile_list = list(self.assets) #chuyen tu dict sang list chi giu lai key la gia tri
self.tile_group = 0 #key của assets
self.tile_variant = 0 # chủng loại của từng thứ của 1 key trong assets

self.clicking = False
self.right_clicking = False
self.shift = False

self.ongrid = True
```

- **self.movement = [False, False, False, False]:** Khởi tạo một danh sách các giá trị boolean đại diện cho việc di chuyển theo bốn hướng, có lẽ để điều khiển việc di chuyển của người chơi.
- **self.tilemap = Tilemap(self, tile_size=16):** Khởi tạo một đối tượng **Tilemap**, có thể để quản lý và hiển thị một bản đồ được tạo từ các ô lưới. Tham số **tile_size** gợi ý về kích thước của các ô lưới.
- Khối **try** với **except FileNotFoundError**: Cố gắng tải một tập tin bản đồ có tên **'map.json'** bằng phương thức load của đối tượng **Tilemap**. Nếu tập tin không được tìm thấy, ngoại lệ được bắt và mã tiếp tục chạy.
- **self.scroll = [0, 0]:** Khởi tạo vị trí cuộn của máy ảnh ở góc trên bên trái, gợi ý rằng việc hiển thị hoặc cửa sổ xem sẽ được cuộn dựa trên các tọa độ này.

- **self.tile_list = list(self.assets):** Chuyển đổi các khóa của một từ điển có tên **assets** thành một danh sách có tên **tile_list**. Điều này có thể hữu ích để lặp qua các khóa hoặc truy cập chúng một cách ngẫu nhiên.
- **self.tile_group = 0**: Khởi tạo một biến **tile_group** có vẻ đại diện cho một khóa từ từ điển **assets**.
- **self.tile_variant = 0:** Khởi tạo một biến **tile_variant** có vẻ đại diện cho một biến thể hoặc dạng con của ô lưới được chỉ định bởi **tile_group**.
- **self.clicking = False, self.right_clicking = False**: Khởi tạo các biến boolean **clicking** và **right_clicking** để đại diện cho trạng thái của các cú nhấp chuột.
- **self.shift = False**: Khởi tạo một biến boolean **shift** để đại diện cho trạng thái của phím Shift.
- **self.ongrid = True**: Khởi tạo một biến boolean **ongrid**, có thể chỉ ra liệu một đối tượng hiện đang được đặt trên một ô lưới hay không.

```
def run(self):
    while True:
        self.display.fill((0, 0, 0))

        self.scroll[0] += (self.movement[1] - self.movement[0]) * 2
        self.scroll[1] += (self.movement[3] - self.movement[2]) * 2

        render_scroll = (int(self.scroll[0]), int(self.scroll[1]))

        self.tilemap.render(self.display, offset=render_scroll)

        current_tile_img = self.assets[self.tile_list[self.tile_group]][self.tile_variant].copy()
        current_tile_img.set_alpha(150)

        mpos = pygame.mouse.get_pos()
        mpos = (mpos[0] / RENDER_SCALE, mpos[1] / RENDER_SCALE)

        tile_pos = (int((mpos[0] + self.scroll[0]) // self.tilemap.tile_size), int((mpos[1] + self.scroll[1]) // self.tilemap.tile_size))
```

phương thức run() định nghĩa một vòng lặp chạy liên tục để cập nhật và vẽ các phần tử của trò chơi hoặc ứng dụng đồ họa.

- **self.display.fill((0, 0, 0)):** Xóa màn hình bằng cách tô đen (**(0, 0, 0)** là mã màu đen).
- **self.scroll[0] += (self.movement[1] - self.movement[0]) * 2 và self.scroll[1] += (self.movement[3] - self.movement[2]) * 2:** Cập nhật vị trí cuộn của camera dựa trên trạng thái của biến self.movement. Có vẻ như nếu **self.movement[0]** được kích hoạt (đặt thành **True**), camera sẽ di chuyển sang trái, và nếu **self.movement[1]** được kích hoạt, camera sẽ di chuyển sang phải. Tương tự, **self.movement[2]** và **self.movement[3]** điều khiển di chuyển lên và xuống của camera.
- **render_scroll = (int(self.scroll[0]), int(self.scroll[1])):** Chuyển đổi vị trí cuộn của camera thành kiểu dữ liệu nguyên (integer) để sử dụng khi vẽ các phần tử trên màn hình.
- **self.tilemap.render(self.display, offset=render_scroll):** Vẽ bản đồ được đại diện bởi **self.tilemap** lên màn hình, sử dụng **render_scroll** để xác định vị trí của camera.
- **current_tile_img = self.assets[self.tile_list[self.tile_group]][self.tile_variant].copy():** Tạo một bản sao

của hình ảnh tile hiện tại từ danh sách tài nguyên **assets**. Tile cụ thể được chọn bằng cách sử dụng **self.tile_list[self.tile_group]** và **self.tile_variant**.

- **current_tile_img.set_alpha(150):** Đặt độ mờ (alpha) của hình ảnh tile hiện tại thành 150. Điều này có thể làm cho tile trở nên mờ hơn khi hiển thị trên màn hình.
- **mpos = pygame.mouse.get_pos():** Lấy vị trí của con trỏ chuột.
- **mpos = (mpos[0] / RENDER_SCALE, mpos[1] / RENDER_SCALE):** Chia vị trí chuột cho một hằng số **RENDER_SCALE**, có thể để thích nghi với tỉ lệ hiển thị của màn hình.
- **tile_pos = (int((mpos[0] + self.scroll[0]) // self.tilemap.tile_size), int((mpos[1] + self.scroll[1]) // self.tilemap.tile_size)):** Xác định vị trí của tile dưới con trỏ chuột trên bản đồ. Điều này bằng cách tính toán tọa độ ô lưới tương ứng với vị trí con trỏ chuột.

```
if self.ongrid:
    self.display.blit(current_tile_img, (tile_pos[0] * self.tilemap.tile_size - self.scroll[0], tile_pos[1] * self.tilemap.tile_size - self.scroll[1]))
else:
    self.display.blit(current_tile_img, mpos)

if self.clicking and self.ongrid:
    self.tilemap.tilemap[str(tile_pos[0]) + ';' + str(tile_pos[1])] = {'type': self.tile_list[self.tile_group], 'variant': self.tile_variant, 'pos': tile_pos}

if self.right_clicking:
    tile_loc = str(tile_pos[0]) + ';' + str(tile_pos[1])
    if tile_loc in self.tilemap.tilemap:
        del self.tilemap.tilemap[tile_loc]
    for tile in self.tilemap.offgrid_tiles.copy():
        tile_img = self.assets[tile['type']][tile['variant']]
        tile_r = pygame.Rect(tile['pos'][0] - self.scroll[0], tile['pos'][1] - self.scroll[1], tile_img.get_width(), tile_img.get_height())
        if tile_r.collidepoint(mpos):
            self.tilemap.offgrid_tiles.remove(tile)
```

Đoạn code chứa một loạt các điều kiện để xử lý hành động của người chơi khi họ tương tác với bản đồ hoặc các tile.

- **if self.ongrid::** Kiểm tra xem người chơi có đang tương tác với lưới ô không. Nếu có, các hành động sau sẽ thực hiện trên lưới ô.
- **self.display.blit(current_tile_img, (tile_pos[0] * self.tilemap.tile_size - self.scroll[0], tile_pos[1] * self.tilemap.tile_size - self.scroll[1])):** Vẽ hình ảnh tile hiện tại lên màn hình tại vị trí của con trỏ chuột trên lưới ô. Nếu **self.ongrid** là True, tile sẽ được vẽ tại vị trí của ô lưới đó.
- **else::** Nếu người chơi không đang tương tác với lưới ô, các hành động sau sẽ thực hiện ngoài lưới ô.
- **self.display.blit(current_tile_img, mpos):** Vẽ hình ảnh tile hiện tại lên màn hình tại vị trí của con trỏ chuột. Nếu **self.ongrid** là False, tile sẽ được vẽ tại vị trí của con trỏ chuột.
- **if self.clicking and self.ongrid::** Kiểm tra xem người chơi có đang nhấn chuột trái và có đang tương tác với lưới ô không. Nếu có, thêm thông tin về tile vào bản đồ tile của **self.tilemap** tại vị trí con trỏ chuột trên lưới ô.
- **if self.right_clicking::** Kiểm tra xem người chơi có đang nhấn chuột phải không. Nếu có, thực hiện các hành động sau.

- **tile_loc = str(tile_pos[0]) + ';' + str(tile_pos[1]):** Xác định vị trí của tile trên lưới ô dưới con trỏ chuột.
- **if tile_loc in self.tilemap.tilemap::** Kiểm tra xem vị trí tile trên lưới ô có tồn tại trong bản đồ tile của **self.tilemap** không. Nếu có, thực hiện các hành động sau.
- **del self.tilemap.tilemap[tile_loc]:** Xóa thông tin của tile tại vị trí đó ra khỏi bản đồ tile của self.tilemap.
- **for tile in self.tilemap.offgrid_tiles.copy()::** Duyệt qua tất cả các tile ngoài lưới ô trong **self.tilemap.offgrid_tiles.**
- **tile_img = self.assets[tile['type']][tile['variant']**]]: Lấy hình ảnh của tile từ danh sách tài nguyên self.assets.
- **tile_r = pygame.Rect(tile['pos'][0] - self.scroll[0], tile['pos'][1] - self.scroll[1], tile_img.get_width(), tile_img.get_height()):** Tạo một hình chữ nhật đại diện cho tile này trên màn hình.
- **if tile_r.collidepoint(mpos)::** Kiểm tra xem vị trí con trỏ chuột có nằm trong hình chữ nhật của tile hay không.
- **self.tilemap.offgrid_tiles.remove(tile):** Nếu có, xóa tile đó ra khỏi danh sách các tile ngoài lưới ô

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:   # Click x on window
        pygame.quit()
        sys.exit()

    if event.type == pygame.MOUSEBUTTONDOWN:
        if event.button == 1:
            self.clicking = True
            if not self.ongrid:
                self.tilemap.offgrid_tiles.append({'type': self.tile_list[self.tile_group], 'variant': self.tile_variant, 'pos': (mpos[0] + self.scroll[0], mpos[1] + self.scroll[1])})
        if event.button == 3:
            self.right_clicking = True
        if self.shift:
            if event.button == 4:
                self.tile_variant = (self.tile_variant - 1) % len(self.assets[self.tile_list[self.tile_group]])
                #trick lỏ để lập lại từ đầu(giả sử có 4 ptu thì lập từ 0 1 2 3 rồi về 0 do đ' có vụ phần dư như thế)
            if event.button == 5:
                self.tile_variant = (self.tile_variant + 1) % len(self.assets[self.tile_list[self.tile_group]])
        else:
            if event.button == 4:
                self.tile_group = (self.tile_group - 1) % len(self.tile_list)
                self.tile_variant = 0
            if event.button == 5:
                self.tile_group = (self.tile_group + 1) % len(self.tile_list)
                self.tile_variant = 0
```

Dòng mã này thực hiện một số hành động khi người chơi tương tác với cửa sổ trò chơi hoặc với chuột.

- **self.display.blit(current_tile_img, (5, 5)):** Vẽ hình ảnh của tile hiện tại lên màn hình tại vị trí cụ thể (5, 5). Điều này có thể là để hiển thị một góc nhỏ của tile nào đó trên màn hình, có thể là cho mục đích thử nghiệm hoặc hiển thị thông tin.
- **for event in pygame.event.get()::** Duyệt qua tất cả các sự kiện trong hàng đợi của pygame.
- **if event.type == pygame.QUIT::** Kiểm tra xem người chơi đã nhấn nút "X" để đóng cửa sổ trò chơi chưa. Nếu có, thoát khỏi trò chơi.

- **if event.type == pygame.MOUSEBUTTONDOWN**:: Kiểm tra xem người chơi đã nhấn chuột chưa.
- **if event.button == 1::** Kiểm tra xem nút chuột nào đã được nhấn. Nếu là nút trái (1), đặt **self.clicking** thành True và thực hiện các hành động sau.
- **if event.button == 3::** Nếu nút chuột là nút phải (3), đặt **self.right_clicking** thành True.
- **if self.shift::** Kiểm tra xem phím Shift có đang được nhấn không. Nếu có, thực hiện các hành động sau.
- **if event.button == 4: và if event.button == 5::** Kiểm tra xem nút lăn lên (4) hoặc nút lăn xuống (5) đã được nhấn. Nếu có, thay đổi **self.tile_variant** hoặc **self.tile_group** tùy thuộc vào phím lăn nào được nhấn.
- Trong các trường hợp khác, thực hiện việc thay đổi **self.tile_variant** hoặc **self.tile_group** dựa trên nút lăn lên hoặc lăn xuống, và thiết lập **self.tile_variant** thành 0 nếu có sự thay đổi trong **self.tile_group.**

```
if event.type == pygame.MOUSEBUTTONUP:
    if event.button == 1:
        self.clicking = False
    if event.button == 3:
        self.right_clicking = False

if event.type == pygame.KEYDOWN:    # dang nhan nut
    if event.key == pygame.K_a:
        self.movement[0] = True
    if event.key == pygame.K_d:
        self.movement[1] = True
    if event.key == pygame.K_w:
        self.movement[2] = True
    if event.key == pygame.K_s:
        self.movement[3] = True
    if event.key == pygame.K_g:
        self.ongrid = not self.ongrid
    if event.key == pygame.K_t:
        self.tilemap.autotile()
    if event.key == pygame.K_o:
        self.tilemap.save('map.json')
    if event.key == pygame.K_LSHIFT:
        self.shift = True

if event.type == pygame.KEYUP:   # tha nut ra
    if event.key == pygame.K_a:
        self.movement[0] = False
    if event.key == pygame.K_d:
        self.movement[1] = False
    if event.key == pygame.K_w:
        self.movement[2] = False
    if event.key == pygame.K_s:
        self.movement[3] = False
    if event.key == pygame.K_LSHIFT:
        self.shift = False
```

Dòng mã này xử lý các sự kiện khi người chơi nhấn và tha nút chuột hoặc bàn phím.

- **if event.type == pygame.MOUSEBUTTONUP::** Kiểm tra xem người chơi đã thả nút chuột chưa.
  - **if event.button == 1:**: Nếu nút chuột là nút trái (1), đặt **self.clicking** thành False.
  - **if event.button == 3::** Nếu nút chuột là nút phải (3), đặt **self.right_clicking** thành False.
- **if event.type == pygame.KEYDOWN::** Kiểm tra xem người chơi đang nhấn một phím nào đó trên bàn phím.
  - Dùng các điều kiện để kiểm tra xem phím nào đã được nhấn và thực hiện các hành động tương ứng. Ví dụ:

    - Nếu phím "a" được nhấn, đặt **self.movement[0]** thành True để di chuyển sang trái.
    - Nếu phím "d" được nhấn, đặt **self.movement[1]** thành True để di chuyển sang phải.
    - Nếu phím "w" được nhấn, đặt **self.movement[2]** thành True để di chuyển lên.
    - Nếu phím "s" được nhấn, đặt **self.movement[3]** thành True để di chuyển xuống.
    - Nếu phím "g" được nhấn, đảo ngược trạng thái của **self.ongrid** giữa True và False.
    - Nếu phím "t" được nhấn, gọi phương thức **autotile()** của **self.tilemap**.
    - Nếu phím "o" được nhấn, gọi phương thức save(**'map.json'**) của **self.tilemap** để lưu bản đồ.
    - Nếu phím "Left Shift" được nhấn, đặt **self.shift** thành True.
- **if event.type == pygame.KEYUP**:: Kiểm tra xem người chơi đã thả phím đang giữ chưa.
    - Dùng các điều kiện tương tự như ở trên để kiểm tra và đặt giá trị tương ứng của **self.movement** hoặc **self.shift** thành False khi phím được thả ra.

```
self.screen.blit(pygame.transform.scale(self.display, self.screen.get_size()), (0, 0))
pygame.display.update()
self.clock.tick(60)  # 60FPS
```

Dòng mã này là phần kết thúc của vòng lặp trong phương thức run(), và có nhiệm vụ vẽ nội dung lên màn hình và cập nhật màn hình.

- **self.screen.blit(pygame.transform.scale(self.display, self.screen.get_size()), (0, 0)):** Dòng này vẽ nội dung từ bề mặt **self.display** (nơi mà tất cả các phần tử trò chơi đã được vẽ) lên bề mặt **self.screen** (bề mặt của cửa sổ trò chơi).
    - **pygame.transform.scale(self.display, self.screen.get_size()):** Đầu tiên, hình ảnh trên **self.display** được co dãn hoặc thu nhỏ để phù hợp với kích thước của cửa sổ trò chơi (**self.screen**).
    - **self.screen.blit(...):** Sau đó, hình ảnh đã được co dãn hoặc thu nhỏ này được vẽ lên bề mặt của cửa sổ trò chơi.

- **pygame.display.update():** Dòng này cập nhật màn hình để hiển thị nội dung mới đã được vẽ lên màn hình. Điều này là cần thiết để người chơi có thể thấy các thay đổi trên màn hình.
- **self.clock.tick(60):** Điều này giới hạn số lần vòng lặp được thực hiện mỗi giây (FPS) thành 60 lần. Việc này giữ cho trò chơi chạy ở tốc độ ổn định và tránh quá nhiều tài nguyên được sử dụng.

# PHẦN IV: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Sau khi thực thi và hoàn thành ứng dụng game chiến đấu 2D, nhóm chúng em đã rút ra được một số kết luận và tổng hợp hướng phát triển tương lai cho ứng dụng như sau:

1. **Kết luận:**

**1.1. Thành tựu:**

- Đã tạo ra một game chiến đấu 2D với đồ họa hấp dẫn và hệ thống chiến đấu đa dạng.
- Phát triển các nhân vật và môi trường chơi đa dạng, mang lại trải nghiệm độc đáo cho người chơi.
- Tạo ra một sản phẩm giải trí hấp dẫn, phù hợp cho nhiều đối tượng người chơi.

**1.2. Hạn chế và cơ hội cải thiện:**

- Có thể gặp phải các lỗi và vấn đề kỹ thuật trong quá trình thử nghiệm và sử dụng.
- Giao diện người dùng cần được cải thiện để tăng tính thẩm mỹ và trải nghiệm người chơi.
- Cần thêm tính năng bổ sung như chế độ chơi mới, các nhân vật và kỹ năng mới để làm giàu nội dung và tăng độ hấp dẫn của game.

**1.3. Hướng phát triển:**

- **Kiểm thử và sửa lỗi:**
- Tiếp tục kiểm thử để xác định và sửa chữa các lỗi, đảm bảo tính ổn định của game.
- **Cải thiện giao diện người dùng:**
- Tối ưu hóa giao diện người dùng để tạo ra trải nghiệm chơi game thú vị và thu hút hơn.
- **Mở rộng nội dung:**
- Phát triển thêm các chế độ chơi mới như chế độ câu chuyện, chế độ trực tuyến để tạo ra trải nghiệm đa dạng cho người chơi.
- Thêm các nhân vật, vũ khí, kỹ năng mới để mở rộng nội dung và độ phong phú của game.
- **Tích hợp âm thanh và hiệu ứng đặc biệt:**
- Tạo ra âm thanh và hiệu ứng đặc biệt phù hợp để tăng cường không gian trải nghiệm của game.
- **Phản hồi từ người chơi:**
- Lắng nghe và xem xét các ý kiến đóng góp từ người chơi để điều chỉnh và cải thiện game theo hướng phù hợp nhất.

Tổng quát, việc liên tục cải thiện và phát triển game chiến đấu 2D là quan trọng để thu hút và giữ chân người chơi, đồng thời tạo ra một sản phẩm giải trí đáng giá trên thị trường game đầy cạnh tranh. Trong quá trình thực hiện làm ứng dụng không tránh khỏi những thiếu sót và hạn do thiếu kinh nghiệm và kiến thức, chúng em kinh mong nhận được những ý kiến đóng góp của thầy để đồ án của chúng em được hoàn thiện hơn.

## TÀI LIỆU THAM KHẢO


1. Kênh youtube tham khảo code

https://www.youtube.com/watch?v=xxRhvyZXd8I&list=PLX5fBCkxJmm1fPSqgn9gyR3qih8yYLvMj&pp=iAQB

2. Trang chủ học code python

Python Tutorial (w3schools.com)

3. Các trang tham khảo khác

Làm game cơ bản với Pygame - MIM Python

Các Yếu Tố Lập Trình Game Cơ Bản Với Pygame (Phần 2) (codelearn.io)