Instituto Politécnico do Cávado e do Ave (IPCA)

Licenciatura em Engenharia e Desenvolvimento de Jogos Digitais

Sistema de controlo por visão

Introdução à visão por computador

2025/2026

Anthony Frade - 31477

Valezka Naia - 31481

Carlos Pereira – 7506

Prof. José Brito

Barcelos, outubro de 2025

Índice

2. Introdução

Este trabalho prático foi realizado no âmbito da unidade curricular de Introdução à Visão por Computador, integrada no curso de Engenharia e Desenvolvimento de Videojogos do Instituto Politécnico do Cávado e do Ave (IPCA).

O objetivo principal do projeto foi familiarizar-se com e aplicar técnicas de visão por computador, com o propósito de controlar um jogo através de uma câmara. Para isso, foram desenvolvidas duas versões funcionais: uma baseada no jogo Space Invaders, criado em Python com a biblioteca pygame e disponibilizado pelo utilizador educ8s no GitHub, e outra inspirada no jogo Super Mario, cujo código foi partilhado pelo utilizador mx0c na mesma plataforma. Ambos os códigos apresentam uma estrutura e uma lógica de integração com o módulo de visão por computador bastante semelhantes, diferenciando-se apenas por pequenas adaptações específicas de cada *gameplay*.

O sistema implementado baseia-se na deteção e no rastreamento de um objeto por cor, cuja posição é utilizada para controlar o movimento da personagem (ou nave) no jogo, substituindo as entradas tradicionais do teclado. Desta forma, a visão por computador funciona como um meio alternativo e interativo de controlo, demonstrando o seu potencial em aplicações de jogos.

3. Descrição do jogo

O sistema em análise integra um módulo de interação por visão computacional, desenvolvido no âmbito deste trabalho, a um motor de jogo pré-existente. Esta arquitetura visa demonstrar a aplicação prática de técnicas de visão por computador como método alternativo de interação em videojogos.

3.1. O motor de jogo: plataforma base

Como supramencionado, o jogo escolhido como plataforma de teste é uma versão do clássico Space Invaders, originalmente desenvolvido em Python com a biblioteca pygame e disponibilizado publicamente pelo utilizador educ8s no GitHub.

3.1.1. Estrutura e mecânicas

O motor de jogo fornece a estrutura fundamental, que inclui:

- A estrutura fixa do *shooter*, na qual uma nave controlada pelo jogador se move horizontalmente na base do ecrã para eliminar inimigos;
- As mecânicas básicas de jogo, como o movimento horizontal, o disparo de *lasers*, a gestão da movimentação e dos padrões dos inimigos alienígenas e a aparição de entidades bónus;
- O ciclo de eventos e a atualização do pygame, que geram a lógica e a renderização (pontuação, vidas, *sprites*), funcionam como o *host* temporal do módulo de controlo.

```
while True:
    # ... (código do OpenCV)

    for event in pygame.event.get():
    # ... (processamento de eventos)

    if game.run:
    # ... (lógica de atualização do jogo)

    # ... (código de renderização)

    pygame.display.update()
    clock.tick(60)
```

Figura 1 - Ciclo de eventos e atualizações do pygame.

Esta componente é tratada como um sistema fechado e funcional, servindo como ambiente de execução em que os comandos gerados pelo módulo de visão são executados.

3.2. Módulo de interação por visão de computador

O centro deste projeto reside no desenvolvimento e na implementação do sistema de controlo por visão, que substitui o tradicional *input* por teclado. Este módulo utiliza a biblioteca OpenCV (*Open Source Computer Vision*) para transformar dados visuais em comandos de jogo.

3.2.1. O paradigma do controlo por visão

O controlo é executado por meio da deteção e do rastreamento de um objeto físico pela câmara de cor pré-especificada. A lógica implementada formaliza o controlo como uma função de mapeamento discreto:

$f: I \rightarrow C$

I: Sequência de *frames* de vídeo (imagens matriciais)

C: Vetor de comandos de jogo (*isMovingLeft, isShooting* e *isMovingRight)*.

3.2.2 Código do OpenCV

A implementação destas funções segue um código otimizado para a operação em tempo real:

- Aquisição e segmentação: captura de *frames* e processamento imediato, por exemplo, filtragem por cores no espaço de cor *HSV*, para isolar o objeto de interesse;

```
ret, frame = cap.read()
if ret == True:
    frame_mirror = frame[:, ::-1, :]
    image_HSV = cv2.cvtColor(frame_mirror, cv2.COLOR_BGR2HSV)
    # ... (criação de máscaras e limiares)
    image_thresholded = image_thresholded_h * image_thresholded_s * image_thresholded_v
```

Figura 2 - Aquisição e segmentação.

- Extração de *contours*: o *contour* maior é extraído e calcula-se o centro de massa (x, y) do objeto segmentado, que representa sua posição.

```
contours, hierarchy = cv2.findContours(image=image_thresholded,
                                       mode=cv2.RETR_EXTERNAL,
                                       method=cv2.CHAIN_APPROX_NONE)
# ... (encontra o maior contorno)

if i == i_max_area:
    # ...
    M = cv2.moments(array=contour)
    if M['m00'] != 0:
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])
```

Figura 3 - Extração de *contours.*

- Lógica da decisão e mapeamento de comandos: a posição horizontal do centro determina a direção de movimento da nave, utilizando uma zona morta central para garantir estabilidade e precisão.

```
# Limite horizontal para deslocar para a esquerda/direita
center_zone = frame_width // 3

if cXCopy != -100:
    # Mover para a esquerda
    if cXCopy < frame_width // 2 - center_zone // 2:
        isMovingLeft = True
    # Mover para a direita
    elif cXCopy > frame_width // 2 + center_zone // 2:
        isMovingRight = True

    # Disparo
    if cYCopy < frame_height // 3:
        isShooting = True
```

Figura 4 - Lógica da decisão e mapeamento dos comandos.

A posição vertical do objeto (nomeadamente, a sua presença na zona superior do campo de visão) é mapeada para a ação de disparo.

4. Metodologia de implementação e análise técnica

Esta secção detalha a implementação do módulo de interação por visão de computador, descrevendo a metodologia técnica aplicada a cada etapa do código.

4.1. Aquisição, calibração e pré-processamento

4.1.1. Aquisição de *frames* e reflexão horizontal

O vídeo da câmara é capturado sequencialmente através do método *cv2.VideoCapture*. O passo inicial de pré-processamento é a reflexão horizontal do *frame:*

Esta operação de *slicing* em matrizes (numPy) inverte a ordem do eixo da largura, garantindo a correspondência intuitiva do movimento: mover o objeto para a esquerda no mundo real move a nave para a esquerda no ecrã.

```
cap = cv2.VideoCapture()

while True:
    if not (cap.isOpened()):
        cap.open(0)
    ret, frame = cap.read()

    if ret == True:
        # inverte horizontalmente ("espelho")
        frame_mirror = frame[:, ::-1, :]
```

Figura 5 - Aquisição dos *frames* e reflexão horizontal.

A escolha de refletir horizontalmente os *frames* na etapa inicial de pré-processamento foi feita para estabelecer uma correspondência direta entre os movimentos do utilizador e as ações da nave no ecrã. Se esta inversão não fosse feita, o controlo tornar-se-ia contraintuitivo, exigindo uma adaptação que não seria necessária do utilizador e aumentando a probabilidade de cometer erros na interação. Assim, esta decisão visa melhorar a ergonomia e a experiência do utilizador, tornando o sistema mais natural e fácil de usar.

4.1.2 Calibração dinâmica *threshold* (HSV)

Para manter a funcionalidade diante das várias variações de iluminação, a segmentação cromática é suportada por uma interface de calibração dinâmica. Utilizam-se os *trackbars* do OpenCV (*cv2.createTrackbar*) para permitir o ajuste em tempo real dos parâmetros de

threshold (mínimo e máximo de H, S e V). Esta abordagem evita o uso de valores fixos (*hardcoding*), o que aumenta a adaptabilidade do sistema.

```
cv2.namedWindow("imageshow")
cv2.createTrackbar("H MIN", "imageshow", threshhold_h_min, 180, on_trackbar_change_h_min)
cv2.createTrackbar("H MAX", "imageshow", threshhold_h_max, 180, on_trackbar_change_h_max)
cv2.createTrackbar("S MIN", "imageshow", threshhold_s_min, 255, on_trackbar_change_s_min)
cv2.createTrackbar("S MAX", "imageshow", threshhold_s_max, 255, on_trackbar_change_s_max)
cv2.createTrackbar("V MIN", "imageshow", threshhold_v_min, 255, on_trackbar_change_v_min)
cv2.createTrackbar("V MAX", "imageshow", threshhold_v_max, 255, on_trackbar_change_v_max)
```

Figura 6 - Calibração dinâmica *threshold* (HSV).

A calibração dinâmica do *threshold* no espaço HSV foi implementada para tornar o sistema mais robusto, especialmente diante de variações de iluminação e diferenças entre câmaras. Usar valores fixos podia causar problemas na segmentação da cor do objeto em ambientes distintos. Com os *trackbars* do OpenCV, podemos fazer ajustes em tempo real, o que nos dá flexibilidade e ajuda a manter um bom desempenho do sistema, sem precisarmos reescrever o código ou fazer alterações complicadas.

4.2 Segmentação cromática e extração de descritores

4.2.1 Conversão para HSV e *threshold* granular

O *frame* é convertido para o espaço de cores HSV (*cv2.cvtColor*) devido à sua maior invariância à iluminância. Em seguida, realiza-se a segmentação cromática através da definição de intervalos para cada canal (H, S e V), de acordo com os limites estabelecidos durante a calibração dinâmica:

$$\begin{cases} Hmin \leq H \leq Hmax \\ Smin \leq S \leq Smax \\ Vmin \leq V \leq Vmax \end{cases}$$

Isto determina o intervalo de valores correspondentes à cor do objeto a ser rastreado. Com base neles, é criada uma máscara binária que indica, para cada *pixel*, se este pertence ou não à cor de interesse:

$$M_{(x,y)} = \begin{cases} 1 \; se \; Hmin \leq H_{(x,y)} \leq Hmax \wedge Smin \leq S_{(x,y)} \leq Smax \wedge Vmin \leq V_{(x,y)} \leq Vmax \\ 0 \; caso \; contrário \end{cases}$$

A composição lógica das máscaras é realizada por meio da multiplicação de matrizes (equivalente à operação *AND*), o que permite o isolamento preciso da cor do objeto.

```
image_HSV = cv2.cvtColor(frame_mirror, cv2.COLOR_BGR2HSV)

# ... (código de threshold para cada canal H, S, V)

# máscara final em preto e branco, o objeto desejado é branco
image_thresholded = image_thresholded_h * image_thresholded_s * image_thresholded_v
```

Figura 7 - Conversão para HSV e *threshold*.

Optámos por utilizar o espaço de cores HSV para a segmentação cromática, devido à sua maior resistência às variações de iluminação. Isto torna a deteção da cor-alvo mais fiável. Ao aplicar um *threshold* granular em cada canal e combinar as máscaras de forma lógica, conseguimos uma segmentação mais precisa, minimizando a interferência de ruído ou de cores semelhantes no fundo. Esta abordagem encontra um bom equilíbrio entre simplicidade computacional e precisão, sendo adequada para aplicações interativas em tempo real.

4.2.2 Extração de contornos e análise morfológica

A máscara binária é analisada para extração de contornos por meio de *cv2.findContours*.

```
# devolve apenas os contours externos
contours, hierarchy = cv2.findContours(image=image_thresholded,
                                       mode=cv2.RETR_EXTERNAL,  # apenas contornos externos
                                       method=cv2.CHAIN_APPROX_NONE)  # guarda todos os pontos do contorno
```

Figura 8 - Extração de *contours*.

A parametrização utilizada é crucial para a eficiência:

- *mode=cv2.RETR_EXTERNAL*: parâmetro que instrui o algoritmo a detectar apenas os contornos externos, ignorando os internos. Esta opção reduz o processamento desnecessário e concentra a análise nas bordas principais do objeto de interesse;

- *method=cv2.CHAIN_APPROX_NONE*: preserva todos os pontos do *contour* para garantir a máxima precisão possível no cálculo de descritores subsequentes.

Na extração de contornos, a configuração escolhida (*cv2.RETR_EXTERNAL* e *cv2.CHAIN_APPROX_NONE*) foi concebida para equilibrar eficiência e precisão. Ao focar apenas nos contornos externos, evitamos detalhes desnecessários e reduzimos a carga computacional. Manter todos os pontos do contorno assegura que os descritores, como o centro de massa, sejam precisos. Isto é fundamental para interpretar corretamente os movimentos do objeto e traduzi-los em comandos do jogo.

4.2.3 Cálculo do centro

Para quantificar a posição do objeto, calculam-se os seus momentos de imagem. Primeiramente, apenas o contorno com a área máxima (M00) é selecionado para filtrar o ruído. O centro de massa (*cx, cy*) é então derivado dos momentos de ordem 0 e 1:

O código inclui uma verificação crucial para evitar divisão por zero caso nenhum objeto seja detetado.

```python
# encontrar o índice e a área do maior objeto
i_max_area = -1
max_area = 0

for i in range(len(contours)):
    contour = contours[i]
    c_area = cv2.contourArea(contour=contour)
    if c_area > max_area:
        max_area = c_area
        i_max_area = i

# ... (dentro do loop que desenha o maior contorno)
M = cv2.moments(array=contour)

# Verifica se o m00 é diferente de 0
# para não dividir por zero
if M['m00'] != 0:
    cX = int(M["m10"] / M["m00"])
    cY = int(M["m01"] / M["m00"])
```

Figura 9 - Maior *contour* é identificado e usado para calcular o centro de massa.

4.3 Mapeamento para os comandos do jogo

A posição do centro de massa (*cx, cy*) é traduzida em um vetor de comandos binários (*isMovingLeft*, *isShooting*, *isMovingRight)* que serão consumidos pelo motor de jogo:

- Controlo horizontal: É aplicada uma histerese espacial, ou "zona morta" (*center_zone*), no centro do ecrã. O movimento só é ativado se o objeto cruzar os limites desta zona, garantindo a estabilidade da nave e prevenindo oscilações indesejáveis (*jitter*).
- Controlo do disparo: O comando de disparo é mapeado para a entrada do objeto no terço superior do ecrã, o que define um gesto vertical claro e intencional.

```
# definir uma zona neutra para que quando o objeto está no meio, a nave não se move
center_zone = frame_width // 3  # zona central "morta"

# ...

if cXCopy != -100:
    # Mover para a esquerda se o centro do objeto (cXCopy) está à esquerda da zona central
    if cXCopy < frame_width // 2 - center_zone // 2:
        isMovingLeft = True
    # Mover para a direita se o centro do objeto (cXCopy) está à direita da zona central
    elif cXCopy > frame_width // 2 + center_zone // 2:
        isMovingRight = True

    # Disparo quando o objeto vai para cima (menor Y → mais alto)
    if cYCopy < frame_height // 3:
        isShooting = True
```

Figura 10 - A posição do centro é mapeada aos comandos de movimento e disparo.

$$f(cx, cy) = \begin{cases} isMovingLeft = 1 \ \ se \ \ cx < frame\_width//2 - center\_zone//2 \\ isMovingRigth = 1 \ se \ cx > frame\_width//2 + center\_zone//2 \\ isShooting = 1 \ se \ cy \ \ < frame\_height//3 \end{cases}$$

Por exemplo, se a imagem tiver 600 px de largura o *center_zone* seria 200.

Isto significa:

- Se *cXCopy* < 200 então o objeto está à esquerda e a nave se ira mover para a esquerda;
- Se *cXCopy* > 400 então o objeto está à direita e a nave se ira mover para a direita;
- Entre 200 e 400 seria a zona central, a nave não se move.

As equações *frame_width//2 – center_zone//2 e frame_width//2 + center_zone//2* definirão as zonas à esquerda, á direita e na zona central, onde a nave não se mexe.

Durante o desenvolvimento desta fase, começámos por explorar o uso da biblioteca pyautogui para simular comandos de teclado com base nas posições detetadas pelo OpenCV. No entanto, acabámos por abandonar essa abordagem devido a problemas de desempenho e de fiabilidade. A pyautogui introduzia uma latência visível e precisava de focos específicos na janela, o que dificultava a sincronização entre a deteção do movimento e a atualização do jogo. Além disso, o controlo com pyautogui não permitia implementar facilmente zonas de ação morta ou limites espaciais, que são importantes para evitar movimentos involuntários da nave. Por isso, optámos por criar um vetor de comandos que o motor do pygame pudesse usar diretamente, garantindo uma resposta rápida, estável e integrada ao ciclo principal do jogo.

4.4 Integração síncrona com pygame

A latência é minimizada pelo acoplamento síncrono entre os módulos. O OpenCV é executado em cada iteração do ciclo principal do pygame. O vetor de comandos gerado é consumido imediatamente na fase de atualização do estado do jogo. Esta sincronização garante que as ações de movimento e disparo refletem o estado visual mais recente, otimizando a experiência em tempo real.

```
# ... (todo o código OpenCV acima)

# Updating
if game.run:
    spaceship = game.spaceship_group.sprite

    # ----- movimento controlado pela câmara -----
    if isMovingLeft:
        spaceship.rect.x -= spaceship.speed
    elif isMovingRight:
        spaceship.rect.x += spaceship.speed

    # ----- disparo -----
    if isShooting and spaceship.laser_ready:
        spaceship.shoot_laser()  # (A função shoot_laser contém a lógica de disparo)

    # ... (restante da lógica de atualização do Pygame)
```

Figura 11 – *"While true" loop.*

Por fim, optámos por usar a integração síncrona com o pygame para reduzir a latência entre a deteção visual e a atualização do estado do jogo. Processar as imagens no ciclo principal garante que cada *frame* capturado seja refletido imediatamente nas ações do jogo, tornando a interação mais fluida. Esta abordagem mantém a consistência entre o que é visto e a resposta do jogo, o que é fundamental para uma experiência de controlo baseada em visão por computador.

5. Análise de desafios e soluções implementadas

Ao longo deste projeto, encontramos vários desafios, porém o maior deles foi descobrir e encontrar informação sobre como traduzir o movimento da câmara para os comandos de jogo e como fazer essa conexão sem inserir bibliotecas externas como pyautogui. Depois de várias tentativas e erros, conseguimos alcançar uma solução viável.

Um dos outros desafios que encontramos foi a elevada sensibilidade do centro de massa, que pode provocar um controlo impreciso da nave, comprometendo a sua estabilidade. Para combater esse problema introduzimos uma zona morta (*center_zone*) para aumentar a estabilidade do sistema. Esta abordagem exige que o desvio do centro de massa ultrapasse um limite predefinido antes de produzir uma resposta, o que previne oscilações em torno do ponto de equilíbrio e promove um movimento mais suave e previsível.

```
# definir uma zona neutra para que quando o objeto está no meio, a nave não se move
# a nave só se vai mover se o objeto estiver claramente à esquerda ou à direita
center_zone = frame_width // 3  # zona central "morta"

# ... (inicialização das variáveis de controlo)

if cXCopy != -100:
    # Mover para a esquerda se o centro do objeto (cXCopy) está à esquerda da zona central
    if cXCopy < frame_width // 2 - center_zone // 2:
        isMovingLeft = True
    # Mover para a direita se o centro do objeto (cXCopy) está à direita da zona central
    elif cXCopy > frame_width // 2 + center_zone // 2:
        isMovingRight = True
    # Se cXCopy estiver dentro da center_zone, isMovingLeft e isMovingRight
    # permanecem False, e a nave não se move.
```

Figura 12 - A zona morta *(center_zone).*

6. Apresentação de resultados e discussão crítica

O sistema implementado mostrou-se funcional, atingindo o objetivo principal de controlar a aplicação em tempo real com latência mínima.

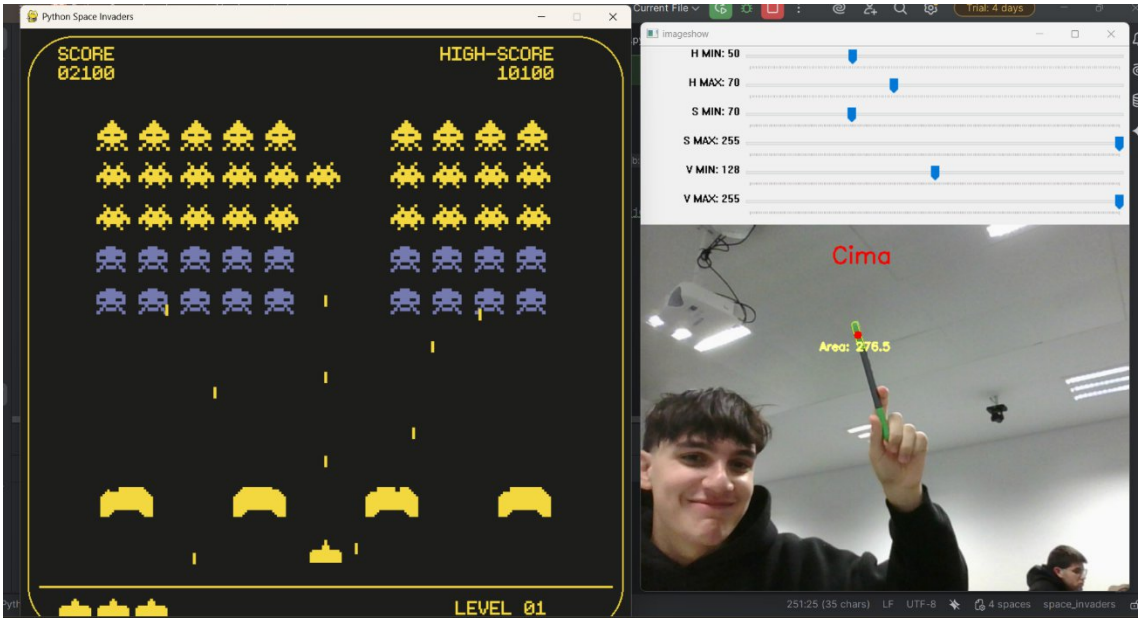

Figura 13 - Jogo com todas as implementações.

6.1. Discussão crítica

A metodologia adotada representa um equilíbrio entre eficiência computacional e robustez semântica.

Entre as vantagens, destacam-se a baixa complexidade dos algoritmos de segmentação por cor e de análise de contornos, que podem ser executados em hardware de baixo custo, dispensando o uso de *GPU* para aceleração.

A implementação de uma calibração dinâmica confere ao sistema uma flexibilidade operacional significativa, possibilitando adaptações em tempo real às variações nas condições de iluminação.

No entanto, o sistema apresenta limitações notáveis, especialmente a sua fragilidade semântica: a performance tende a degradar-se em cenários de oclusão parcial do objeto, de fundos com cores semelhantes ou de mudanças extremas de iluminação.

Além disso, a lógica aplicada limita-se a um único objeto de cor específica, o que dificulta a generalização para múltiplos objetos ou diferentes cores.

6.2. Perspetivas de melhoria

- Segurança na segmentação: utilização de operações morfológicas, como a abertura (*cv2.MORPH_OPEN*), para remover ruído da máscara binária antes de extrair os contornos.
- Estabilidade do rastreamento: implementação de um filtro de Kalman para suavizar as coordenadas do centro de massa e prever a posição do objeto durante breves oclusões.
- Abordagens alternativas: transição para algoritmos de rastreamento que permitam o controlo do jogo com gestos da mão, eliminando assim a necessidade de objetos coloridos.

7. Conclusão

O projeto demonstrou a implementação de visão por computador para o controle de um jogo. A abordagem permitiu a captura, o processamento e a interpretação de informações visuais, usando espaços de cor HSV, operações matriciais e a extração de descritores geométricos a partir de imagens, garantindo precisão na deteção e no rastreamento do objeto de controle.

Foram identificados desafios relevantes, como variações na iluminação, ruído nas máscaras binárias e instabilidade no controle, que foram abordados de forma sistemática. A calibração dinâmica com *trackbars,* a filtragem pelo maior contorno e a implementação de uma zona morta mostraram-se eficazes na mitigação desses problemas, evidenciando a robustez e consistência do sistema.

Este projeto demonstra, de forma clara, o equilíbrio entre eficiência computacional e fiabilidade na aplicação de algoritmos clássicos de visão em sistemas interativos. Além disso, contribui para o fortalecimento dos conhecimentos práticos e teóricos na conceção de interfaces homem-máquina, proporcionando uma base sólida para desenvolvimentos futuros, como melhorias na estabilidade do rastreamento, na riqueza da interação e na incorporação de técnicas de visão mais avançadas.

8. Referências

- Pygame. (n.d.). *pygame.event — Pygame documentation*. Pygame. Recuperado em 31 de outubro de 2025, de https://www.pygame.org/docs/ref/event.html

- Pygame. (n.d.). *pygame.event.post — Pygame documentation*. Pygame. Recuperado em 31 de outubro de 2025, de https://www.pygame.org/docs/ref/event.html#pygame.event.post

- LibreTexts. (n.d.). *3.13: Rect Objects – Making Games with Python and Pygame (Sweigart)*. *LibreTexts*. Recuperado em 31 de outubro de 2025, de https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Making_Games_with_Python_and_Pygame_(Sweigart)/03%3A_Pygame_Basics/3.13%3A_Rect_Objects