



Jogos Interativos Criados Através de Computação Gráfica



Um projeto de desenvolvimento de um Jogo de Xadrez utilizando o PyGame.

O xadrez é um dos jogos de tabuleiro mais antigos e populares do mundo, conhecido por sua complexidade estratégica e tático.

PyChess

Docente: Professor Dr. Joberto S. B. Martins

Sala: Torre Sul, LAB 10.

Horário: Quinta-feira 19:00 - 21:50

E-mail:

joberto.martins@animaeducacao.com.br

Docente: João Vitor do Vale Marques

Sala: Remota - Zoom

Horário: Sexta-feira 19:00 - 21:50

E-mail: vitor@animaeducacao.com.br

Membros

Membros

Aa Nome	# RA
<u>Alfredo Victor do Nascimento Souza Sena</u>	1272019112
<u>Bruno Castelão Sá Barreto</u>	1272117388
<u>Gabriel Antonio Lopes de Castro</u>	1272023100
<u>Gustavo Rafael Vieira Goes</u>	1272117760
<u>João Amaral Lantyer</u>	1271919682
<u>Levi Coutinho Santos</u>	1272116739
<u>Paulo Sérgio Moraes de Oliveira Filho</u>	1272022847



Descrição do Projeto

PyChess é um projeto de desenvolvimento de um Jogo de Xadrez de dois jogadores, proposto pela nossa equipe para aplicar os conhecimentos teóricos da UC de Computação Gráfica e Realidade Virtual. O projeto conta com uma equipe composta por cinco membros. O jogo oferece uma experiência de xadrez com as regras tradicionais e uma interface amigável com dois tabuleiros para cada jogador. Criado na linguagem Python, utilizando da biblioteca PyGame.

Cronograma de reuniões



Este cronograma fornece uma lista completa de reuniões em equipe com seus respectivos tópicos, referentes ao desenvolvimento do projeto.

Cronograma

Aa Nome	Data	Tópico
 <u>Reunião 1</u>	@25 de outubro de 2023	<i>Brainstorm</i> inicial de ideias para o projeto e desenho inicial do tabuleiro e as peças.
 <u>Reunião 2</u>	@28 de novembro de 2023	Adicionando a lógica de seleção de peças.
 <u>Reunião 3</u>	@29 de novembro de 2023	Adicionando a lógica de todas as peças (peão, torre, cavalo, bispo, rei e rainha) e adicionando um segundo tabuleiro para o segundo jogador.
 <u>Reunião 4</u>	@30 de novembro de 2023	Adicionando animação de piscar ao Rei em situação de xeque, organizando o código e documentação.
 <u>Reunião 5</u>	@2 de dezembro de 2023	Adicionando estado de fim de jogo e uma opção de jogar novamente.
 <u>Reunião 6</u>	@7 de dezembro de 2023	Finalizando e organizando o relatório e código.

Levantamento de Requisitos

Requisitos Funcionais:

1. Implementação da movimentação das peças de acordo com as regras tradicionais de xadrez;
2. Possibilidade de ser jogado por dois jogadores, cada um com seu tabuleiro;
3. Demonstrar a peça selecionada e as casas que esta peça pode se mover para com um *highlight*;
4. Mostrar as peças capturadas no canto inferior da tela;
5. O Rei deve piscar ao entrar em xeque;
6. Regra de fim de jogo, identificando o Rei sendo capturado;
7. O jogo deve dar uma opção para podermos jogar novamente.

Requisitos Não Funcionais:

1. Deve ser desenvolvido na linguagem Python utilizando a biblioteca PyGame.
2. Design de interface intuitiva e agradável para o usuário;
3. Boa performance e fluidez do jogo em diferentes plataformas;
4. Compatibilidade com diferentes sistemas operacionais.

Regras de Negócio:

1. Garantir a autenticidade das partidas para evitar trapaças;
2. Facilidade na interação entre jogadores.

Caso de Uso:

1. O usuário inicia o jogo;
2. O usuário movimenta as peças para realizar uma jogada;
3. O jogo identifica as regras, impedindo movimentos inválidos;
4. O jogo identifica o fim da partida (ao rei ser capturado) e da a opção de jogar novamente.

Desenvolvimento do Projeto:



Desenvolvimento



Conclusão



Desenvolvimento

Repositório no GitHub

GitHub - JoaoLantyer/PyChess

Contribute to JoaoLantyer/PyChess development by creating an account on GitHub.

<https://github.com/JoaoLantyer/PyChess>

JoaoLantyer/
PyChess



2 Contributors 0 Issues 0 Stars 1 Fork

O Jogo



Regra do Jogo:

- Cada jogador tem 16 peças: **1 rei, 1 rainha, 2 torres, 2 bispos, 2 cavalos e 8 peões.** O objetivo do jogo é capturar o rei adversário.
- **Movimentos do Peão:** O peão pode se mover para frente e capturar peças na diagonal. Exclusivamente no seu primeiro movimento ele poderá mover duas casas.
- **Movimentos da Torre:** A torre pode se mover quantas casas queira, horizontalmente ou verticalmente sem pular outras peças.
- **Movimentos do Cavalo:** O cavalo tem um padrão de movimento em 'L', podendo pular sobre outras peças.
- **Movimentos do Bispo:** O bispo se move quantas casas queira na diagonal, sem poder ultrapassar suas próprias peças.
- **Movimentos da Rainha:** A rainha combina os movimentos da torre e do bispo, podendo se mover em linha reta e na diagonal por qualquer número de casas desocupadas.
- **Movimentos do Rei:** O rei pode se mover em uma casa em qualquer direção.

Explicação sobre o código

♟ [Explicação Completa do Código do PyChess](#)



Explicação Completa do Código do PyChess



Utilizaremos a linguagem de programação Python.

Adicione os códigos separadamente, seguindo o exemplo do [Google Colab](#) em uma **célula de código**.

```
import pygame  
pygame.init()
```

Primeiramente importamos a biblioteca **pygame** e iniciamos seus módulos através do comando **.init()**.

```
# TELA  
LARG = 1480  
ALT= 740  
tela = pygame.display.set_mode([LARG, ALT])
```

Com o **pygame** iniciado, definimos nossas constantes e variáveis, começamos pela tela:

- **LARG** - Largura da tela do jogo;
- **ALT** - Altura da tela do jogo;
- **tela** - criamos uma janela para exibição com as dimensões especificadas pelas variáveis **LARG** e **ALT**. Utilizamos o comando do **pygame display.set.mode**;

```
# CORES DO TABULEIRO  
cor_quadrados_1_hex = "#70a2a3"  
cor_quadrados_1 = tuple(int(cor_quadrados_1_hex[i:i+2], 16) for i in (1, 3, 5))  
  
cor_quadrados_2_hex = "#b1e4b9"  
cor_quadrados_2 = tuple(int(cor_quadrados_2_hex[i:i+2], 16) for i in (1, 3, 5))
```

Aqui, são definidas as duas cores do tabuleiro em formato hexadecimal. Em seguida, essas cores são convertidas para o formato RGB, que é usado no Pygame. **tuple(int(cor_quadrados_1_hex[i:i+2], 16) for i in (1, 3, 5))** é uma compreensão de lista que pega cada par de caracteres hexadecimais (representando vermelho, verde e azul) e os converte para um valor inteiro.

```
# TEMPO  
relogio = pygame.time.Clock()  
fps = 60  
frame = 0
```

- **relogio** - Criamos um objeto de relógio que pode ser usado para controlar a taxa de atualização do jogo, utilizando a função **time.Clock()** do pygame;
- **fps** - Definimos a taxa de quadros por segundo (**frames per second**) para **60**.
- **frame** - Também criamos uma variável que contará cada quadro, para ser usado em animações de piscar cores.

```
# FIM DO JOGO  
vencedor = ''  
fim_de_jogo = False
```

Aqui determinamos as variáveis relacionadas ao fim de jogo, como:

- **vencedor** - variável para armazenar uma String com a cor do vencedor da partida;
- **fim_de_jogo** - Variável booleana que indica se o jogo acabou ou não.

```
# FONTES
fonte = pygame.font.Font('fonte/Roboto-Bold.ttf', 40)
fonte_fim = pygame.font.Font('fonte/Roboto-Bold.ttf', 80)
fonte_fim_2 = pygame.font.Font('fonte/Roboto-Bold.ttf', 30)
```

Aqui definimos as fontes nos tamanhos adequados para cada um de seus usos. Usaremos as fontes no tamanho 80 e 30 para o banner de fim de jogo.

```
pygame.display.set_caption('PyChess')
```

Definimos o título da janela do aplicativo como **PyChess**, utilizando a função **display.set_caption()** do **pygame**.

```
brancas = ['torre', 'cavalo', 'bispo', 'rainha', 'rei', 'bispo', 'cavalo', 'torre',
          'peao', 'peao', 'peao', 'peao', 'peao', 'peao', 'peao']

brancas_coord = [(0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7),
                  (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]

pretas = ['torre', 'cavalo', 'bispo', 'rainha', 'rei', 'bispo', 'cavalo', 'torre',
          'peao', 'peao', 'peao', 'peao', 'peao', 'peao', 'peao']

pretas_coord = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0),
                 (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]

brancas_capturadas = []
pretas_capturadas = []
```

Arrays de Peças:

- **brancas**: Este array contém strings representando cada uma das peças de xadrez do conjunto branco. A ordem das peças segue a disposição padrão no início de um jogo de xadrez, começando com as peças na linha de trás (torre, cavalo, bispo, rainha, rei, bispo, cavalo, torre) e seguindo com os peões;
- **pretas**: Similar ao array brancas, este array contém as peças do conjunto preto, também na ordem padrão.

Coordenadas das Peças:

- **brancas_coord**: Este array contém tuplas de coordenadas para cada peça no conjunto branco. Cada tupla representa a posição (x, y) de uma peça no tabuleiro. Por exemplo, (0, 7) é a posição da primeira peça (uma torre) no canto inferior esquerdo do tabuleiro;
- **pretas_coord**: Este array contém as coordenadas para as peças pretas, seguindo a mesma lógica. Aqui, (0, 0) representa a posição da primeira peça preta (uma torre) no canto superior esquerdo do tabuleiro.

Listas de Peças Capturadas:

- Também criamos listas vazias que armazenarão as peças capturadas de cada cor: **brancas_capturadas** e **pretas_capturadas**.

```
turso = 0
selecao = 10000
movimentos_validos = []
```

- **turno** - Esta variável é usada para rastrear de quem é o turno atual no jogo. Onde 0 é o turno de seleção da peça branca, e 1 o turno de realização da jogada. Enquanto 2 e 3 representam o turno das peças pretas na mesma ordem. Essa variável é alterada conforme os jogadores fazem seus movimentos, indicando a mudança de turno.
- **selecao** - Utilizaremos desta variável futuramente para armazenar o índice da peça que o jogador da rodada clicou. O número 10000 é só um *placeholder* com um número alto para ter certeza de que está fora do intervalo de índices válidos para as peças no tabuleiro.
- **movimentos_validos** - Esta é uma lista vazia que será usada para armazenar os movimentos válidos para a peça selecionada.

```
tam_normal = 65
tam_pequeno = 35
```

```

# Adicionando a imagem das peças brancas no tamanho normal (Tabuleiro)
torre_branca = pygame.transform.scale(pygame.image.load('pecas/torre_branca.png'), (tam_normal, tam_normal))
cavalo_branca = pygame.transform.scale(pygame.image.load('pecas/cavalo_branca.png'), (tam_normal, tam_normal))
bispo_branca = pygame.transform.scale(pygame.image.load('pecas/bispo_branca.png'), (tam_normal, tam_normal))
rainha_branca = pygame.transform.scale(pygame.image.load('pecas/rainha_branca.png'), (tam_normal, tam_normal))
rei_branca = pygame.transform.scale(pygame.image.load('pecas/rei_branca.png'), (tam_normal, tam_normal))
peao_branca = pygame.transform.scale(pygame.image.load('pecas/peao_branca.png'), (tam_normal, tam_normal))

# Adicionando a imagem das peças brancas no tamanho pequeno (capturadas)
torre_branca_peq = pygame.transform.scale(pygame.image.load('pecas/torre_branca.png'), (tam_pequeno, tam_pequeno))
cavalo_branca_peq = pygame.transform.scale(pygame.image.load('pecas/cavalo_branca.png'), (tam_pequeno, tam_pequeno))
bispo_branca_peq = pygame.transform.scale(pygame.image.load('pecas/bispo_branca.png'), (tam_pequeno, tam_pequeno))
rainha_branca_peq = pygame.transform.scale(pygame.image.load('pecas/rainha_branca.png'), (tam_pequeno, tam_pequeno))
rei_branca_peq = pygame.transform.scale(pygame.image.load('pecas/rei_branca.png'), (tam_pequeno, tam_pequeno))
peao_branca_peq = pygame.transform.scale(pygame.image.load('pecas/peao_branca.png'), (tam_pequeno, tam_pequeno))

# Adicionando a imagem das peças pretas
torre_preta = pygame.transform.scale(pygame.image.load('pecas/torre_preta.png'), (tam_normal, tam_normal))
cavalo_preta = pygame.transform.scale(pygame.image.load('pecas/cavalo_preta.png'), (tam_normal, tam_normal))
bispo_preta = pygame.transform.scale(pygame.image.load('pecas/bispo_preta.png'), (tam_normal, tam_normal))
rainha_preta = pygame.transform.scale(pygame.image.load('pecas/rainha_preta.png'), (tam_normal, tam_normal))
rei_preta = pygame.transform.scale(pygame.image.load('pecas/rei_preta.png'), (tam_normal, tam_normal))
peao_preta = pygame.transform.scale(pygame.image.load('pecas/peao_preta.png'), (tam_normal, tam_normal))

# Adicionando a imagem das peças pretas no tamanho pequeno (capturadas)
torre_preta_peq = pygame.transform.scale(pygame.image.load('pecas/torre_preta.png'), (tam_pequeno, tam_pequeno))
cavalo_preta_peq = pygame.transform.scale(pygame.image.load('pecas/cavalo_preta.png'), (tam_pequeno, tam_pequeno))
bispo_preta_peq = pygame.transform.scale(pygame.image.load('pecas/bispo_preta.png'), (tam_pequeno, tam_pequeno))
rainha_preta_peq = pygame.transform.scale(pygame.image.load('pecas/rainha_preta.png'), (tam_pequeno, tam_pequeno))
rei_preta_peq = pygame.transform.scale(pygame.image.load('pecas/rei_preta.png'), (tam_pequeno, tam_pequeno))
peao_preta_peq = pygame.transform.scale(pygame.image.load('pecas/peao_preta.png'), (tam_pequeno, tam_pequeno))

imagens_branca = [torre_branca, cavalo_branca, bispo_branca, rainha_branca, rei_branca, peao_branca]
imagens_branca_peq = [torre_branca_peq, cavalo_branca_peq, bispo_branca_peq, rainha_branca_peq, rei_branca_peq, peao_branca_peq]

imagens_preta = [torre_preta, cavalo_preta, bispo_preta, rainha_preta, rei_preta, peao_preta]
imagens_preta_peq = [torre_preta_peq, cavalo_preta_peq, bispo_preta_peq, rainha_preta_peq, rei_preta_peq, peao_preta_peq]

pecas = ['torre', 'cavalo', 'bispo', 'rainha', 'rei', 'peao']

```

- **tam_normal = 65:** Esta linha define uma variável **tam_normal** com o valor 65, que será usada para definir o tamanho das imagens das peças de xadrez. O valor 65 representa as dimensões em pixels para a largura e altura das imagens das peças;
- **tam_pequeno = 35:** Aqui fazemos a mesma coisa que fizemos acima, porém com um valor normal, que será usado para desenhar as peças capturadas no canto inferior da tela.

Carregamento e Redimensionamento das Peças:

- **torre_branca, cavalo_branca, bispo_branca, rainha_branca, rei_branca, peao_branca | torre_preta, cavalo_preta, bispo_preta, rainha_preta, rei_preta, peao_preta:** Estas linhas usam **pygame.image.load** para carregar as imagens das peças do xadrez a partir de arquivos PNG. Cada tipo de peça (torre, cavalo, bispo, rainha, rei e peão) tem sua própria imagem;
- Fizemos o mesmo procedimento para as peças menores, utilizando do sufixo **_peq** para elas. Como dito anteriormente, serão usadas para as peças capturadas.
- **pygame.transform.scale:** Este método é usado para redimensionar cada imagem para as dimensões especificadas pela variável **tam_normal** (**e tam_pequeno para as capturadas**). Isso garante que todas as peças tenham um tamanho uniforme no jogo.

Também definimos listas para conter variáveis que referenciam as imagens das peças:

imagens_branca e **imagens_preta**, além de listas para conter as imagens menores com o sufixo **_peq**. E uma lista com strings de todos os nomes das peças do tabuleiro: **pecas**.

```

def pecas_draw_um():
    for i in range(len(brancas)):
        index = pecas.index(brancas[i])
        tela.blit(imagens_branca[index], (brancas_coord[i][0] * 80 + 8, brancas_coord[i][1] * 80 + 8))
        if turno < 2:
            if selecao == i:
                pygame.draw.rect(tela, 'red', [brancas_coord[i][0] * 80, brancas_coord[i][1] * 80, 80, 80], 2)

    for i in range(len(pretas)):
        index = pecas.index(pretas[i])

```

```

tela.blit(imagens_pretas[index], (pretas_coord[i][0] * 80 + 8, pretas_coord[i][1] * 80 + 8))
if turno >= 2:
    if selecao == i:
        pygame.draw.rect(tela, 'red', [pretas_coord[i][0] * 80, pretas_coord[i][1] * 80, 80, 80], 2)

def pecas_draw_dois():

    espelho_brancas_coord = [(7 - x, 7 - y) for x, y in brancas_coord]
    espelho_pretas_coord = [(7 - x, 7 - y) for x, y in pretas_coord]

    for i in range(len(brancas)):
        index = pecas.index(brancas[i])
        tela.blit(imagens_brancas[index], (espelho_brancas_coord[i][0] * 80 + 8 + 840, espelho_brancas_coord[i][1] * 80 + 8))
        if turno < 2:
            if selecao == i:
                pygame.draw.rect(tela, 'red', [espelho_brancas_coord[i][0] * 80 + 840, espelho_brancas_coord[i][1] * 80, 80, 80], 2)

    for i in range(len(pretas)):
        index = pecas.index(pretas[i])
        tela.blit(imagens_pretas[index], (espelho_pretas_coord[i][0] * 80 + 8 + 840, espelho_pretas_coord[i][1] * 80 + 8))
        if turno >= 2:
            if selecao == i:
                pygame.draw.rect(tela, 'red', [espelho_pretas_coord[i][0] * 80 + 840, espelho_pretas_coord[i][1] * 80, 80, 80], 2)

```

- **Função pecas_draw_um** - Esta função desenha as peças brancas e pretas na tela em suas respectivas posições.

- **Desenho das Peças Brancas:**

- O loop **for i in range(len(brancas))** percorre todas as peças brancas.
- **index = pecas.index(brancas[i])** obtém o índice da peça branca atual na lista **pecas**, que é usado para selecionar a imagem correspondente na lista **imagens_brancas**.
- **tela.blit(imagens_brancas[index], (brancas_coord[i][0] * 80 + 8, brancas_coord[i][1] * 80 + 8))** desenha a peça branca na tela. As coordenadas são ajustadas para posicionar corretamente as peças no tabuleiro.
- Se a peça branca atual está selecionada (**selecao == i**) e é o turno das brancas (**turno < 2**), um retângulo vermelho é desenhado ao redor da peça para indicar que ela está selecionada.

- **Desenho das Peças Pretas:**

- Similarmente, o loop para as peças pretas desenha cada peça preta na tela.
Se uma peça preta está selecionada e é o turno das pretas (**turno >= 2**), um retângulo vermelho é desenhado ao redor dela.

- **Função pecas_draw_dois** - Esta função funciona da mesma forma que **pecas_draw_um**, mas desenha as peças no tabuleiro do **jogador 2**. Utilizamos do “**espelhamento**” das peças de forma em que o segundo jogador também consiga jogar com suas peças no canto inferior do tabuleiro, utilizando o segundo tabuleiro. Fizemos isso utilizando das coordenadas **7 - x e 7 - y**, já que ao fazer isso teremos o x e o y na posição oposta do que tínhamos no tabuleiro do primeiro jogador.

```

# Jogador 1
for linha in range(8):
    for coluna in range(8):
        cor = cores[(linha + coluna) % 2]
        #Tabuleiro
        pygame.draw.rect(tela, cor, [coluna * tam_quadrado, linha * tam_quadrado, tam_quadrado, tam_quadrado])

        #Bordas do Tabuleiro
        pygame.draw.rect(tela, 'black', [coluna * tam_quadrado, linha * tam_quadrado, 2, tam_quadrado])
        pygame.draw.rect(tela, 'black', [coluna * tam_quadrado, linha * tam_quadrado, tam_quadrado, 2])
        pygame.draw.rect(tela, 'black', [0, 0, 642, 642], 2)

        #Retângulo inferior
        pygame.draw.rect(tela, 'black', [0, 642, 642, 98], 2)
        pygame.draw.rect(tela, 'white', [2, 642, 638, 96])

# Jogador 2
for linha in range(8):
    for coluna in range(8):
        cor = cores[(linha + coluna) % 2]
        #Tabuleiro
        pygame.draw.rect(tela, cor, [coluna * tam_quadrado + 840, linha * tam_quadrado, tam_quadrado, tam_quadrado])

        #Bordas do Tabuleiro
        pygame.draw.rect(tela, 'black', [coluna * tam_quadrado + 840, linha * tam_quadrado, 2, tam_quadrado])

```

```

pygame.draw.rect(tela, 'black', [coluna * tam_quadrado + 840, linha * tam_quadrado, tam_quadrado, 2])

#Retângulo inferior
pygame.draw.rect(tela, 'black', [840, 0, 640, 642], 2)
pygame.draw.rect(tela, 'black', [840, 640, 640, 100])

```

tabuleiros_draw - Esta função é responsável por desenhar os dois tabuleiros de xadrez na tela do jogo, um para cada jogador.

- **Parâmetros da Função:**

- **tela:** Este é o objeto de tela do Pygame onde o tabuleiro será desenhado;
- **cores:** Uma tupla contendo as cores dos quadrados do tabuleiro. Passaremos as duas cores definidas anteriormente: **cor_quadrados_1** e **cor_quadrados_2**;

- **Desenho do Tabuleiro para o Jogador 1:**

- **Tamanho do Quadrado:** **tam_quadrado = 640 // 8** calcula o tamanho de cada quadrado do tabuleiro. O tabuleiro tem 8 quadrados em cada direção, então o tamanho total do tabuleiro (640 pixels) é dividido por 8.
- **Loop de Desenho:** Dois loops aninhados (**for linha in range(8)** e **for coluna in range(8)**) percorrem cada linha e coluna do tabuleiro.
- **Seleção de Cor:** **cor = cores[(linha + coluna) % 2]** alterna entre as duas cores fornecidas em cores para cada quadrado.
- **Desenho dos Quadrados:** **pygame.draw.rect(tela, cor, [coluna * tam_quadrado, linha * tam_quadrado, tam_quadrado, tam_quadrado])** desenha cada quadrado do tabuleiro.
- **Desenho das Bordas:** As bordas pretas ao redor de cada quadrado são desenhadas para dar uma aparência mais definida ao tabuleiro.
- **Desenho do retângulo inferior:** Na parte inferior do tabuleiro, também desenhamos um retângulo em branco, da mesma cor que as peças do jogador. Nesta parte da tela é onde aparecerá o texto indicando que é a vez do jogador.

- **Desenho do Tabuleiro para o Jogador 2:**

- Este bloco é quase idêntico ao anterior, mas desenha o segundo tabuleiro deslocado horizontalmente por 840 pixels (+**840**) para o segundo jogador. O Desenho do retângulo inferior é em preto, da mesma cor que as peças do jogador.

```

def texto_draw():
    if turno < 2:
        tela.blit(fonte.render("SUA VEZ", True, 'black'), (20, 670))
    else:
        tela.blit(fonte.render("SUA VEZ", True, 'white'), (860, 670))

```

texto_draw - Esta função é responsável por exibir um texto na tela do jogo para indicar de quem é a vez de jogar. Dependendo do turno atual do jogo, uma mensagem escrita **SUA VEZ** aparecerá no canto inferior da tela do jogador atual.

```

def capturadas_draw():
    for i in range(len(pecas_capturadas)):
        peca_capturada = pecas_capturadas[i]
        index = pecas.index(peca_capturada)
        if i < 8:
            tela.blit(imagens_pretas_peq[index], (230 + 50 * i, 650))
        else:
            tela.blit(imagens_pretas_peq[index], (-170 + 50 * i, 690))

    for i in range(len(brancas_capturadas)):
        peca_capturada = brancas_capturadas[i]
        index = pecas.index(peca_capturada)
        if i < 8:
            tela.blit(imagens_brancas_peq[index], (1070 + 50 * i, 650))
        else:
            tela.blit(imagens_brancas_peq[index], (670 + 50 * i, 690))

```

capturadas_draw - Esta função é responsável por desenhar as peças capturadas na tela, utilizando as versões de tamanho reduzido que defini anteriormente. As peças são desenhadas no canto inferior da tela do jogador que capturou. Ou seja, as peças capturadas da cor preta serão desenhadas no canto inferior do tabuleiro do jogador branco e vice versa.

```

def checar_opcoes(pecas, locs, turno):
    lista_movimentos = []
    todos_movimentos = []
    for i in range(len(pecas)):
        loc = locs[i]
        peca = pecas[i]
        if peca == 'peao':
            lista_movimentos = checar_peao(loc, turno)
        elif peca == 'torre':
            lista_movimentos = checar_torre(loc, turno)
        elif peca == 'cavalo':
            lista_movimentos = checar_cavalo(loc, turno)
        elif peca == 'bispo':
            lista_movimentos = checar_bispo(loc, turno)
        elif peca == 'rainha':
            lista_movimentos = checar_rainha(loc, turno)
        elif peca == 'rei':
            lista_movimentos = checar_rei(loc, turno)
        todos_movimentos.append(lista_movimentos)
    return todos_movimentos

```

checar_opcoes - Esta função é projetada para determinar todos os movimentos possíveis para cada peça de xadrez em um dado estado do jogo.

- **Parâmetros da Função:**

- **pecas** - Este parâmetro armazena a lista de peças de cada jogador, as listas **brancas** e **pretas** que contêm cada peça (rei, rainha, torre, etc);
- **locs** - Este parâmetro armazena a lista de localizações de cada peça, as listas '**brancas_coord**' e '**pretas_coord**';
- **turno** - Indica de quem é o turno atual;

- **Lógica da Função:**

- Inicializamos duas listas, **lista_movimentos** para armazenar os movimentos de uma peça em específico e **todos_movimentos** para armazenar o movimento de todas as peças;
- Depois fazemos um loop pelas peças, obtendo a sua localização e tipo, utilizando de **loc** e **peca**, respectivamente;
- Depois verificamos o tipo da peça e chamamos a função específica correspondente para calcular os movimentos possíveis dessa peça. As funções auxiliares como por exemplo **checar_peao**, contêm a lógica para determinar todos os movimentos válidos para cada tipo de peça, com base em suas regras de movimentação no xadrez.
- Os movimentos possíveis para a peça atual são então adicionados à lista **todos_movimentos**.
- Por fim retornamos a lista **todos_movimentos**, uma lista de listas representando todos os movimentos possíveis para cada peça.

```

def checar_peao(posicao, cor):
    lista_movimentos = []
    if cor == 'branca':
        # Mover pra frente 1 casa
        if (posicao[0], posicao[1] - 1) not in pretas_coord and \
            (posicao[0], posicao[1] - 1) not in brancas_coord and posicao[1] > 0:
            lista_movimentos.append((posicao[0], posicao[1] - 1))

        # Mover pra frente 2 casas, caso esteja na posicao inicial
        if (posicao[0], posicao[1] - 2) not in pretas_coord and \
            (posicao[0], posicao[1] - 2) not in brancas_coord and posicao[1] == 6:
            lista_movimentos.append((posicao[0], posicao[1] - 2))

        # Comer uma peca, indo na diagonal
        if (posicao[0] + 1, posicao[1] - 1) in pretas_coord:
            lista_movimentos.append((posicao[0] + 1, posicao[1] - 1))
        if (posicao[0] - 1, posicao[1] - 1) in pretas_coord:
            lista_movimentos.append((posicao[0] - 1, posicao[1] - 1))

    else:
        # Mover pra frente 1 casa
        if (posicao[0], posicao[1] + 1) not in pretas_coord and \
            (posicao[0], posicao[1] + 1) not in brancas_coord and posicao[1] < 7:
            lista_movimentos.append((posicao[0], posicao[1] + 1))

        # Mover pra frente 2 casas, caso esteja na posicao inicial
        if (posicao[0], posicao[1] + 2) not in pretas_coord and \
            (posicao[0], posicao[1] + 2) not in brancas_coord and posicao[1] == 1:

```

```

        lista_movimentos.append((posicao[0], posicao[1] + 2))

    # Comer uma peça, indo na diagonal
    if (posicao[0] + 1, posicao[1] + 1) in brancas_coord:
        lista_movimentos.append((posicao[0] + 1, posicao[1] + 1))
    if (posicao[0] - 1, posicao[1] + 1) in brancas_coord:
        lista_movimentos.append((posicao[0] - 1, posicao[1] + 1))

return lista_movimentos

```

checar_peao - Projetada para calcular todos os movimentos possíveis para o peão:

- **Parâmetros da Função**

posicao: As coordenadas atuais do peão no tabuleiro (dadas como uma tupla, (x, y)).

cor: A cor do peão, que pode ser 'branca' ou 'preta'. Isso é importante porque os peões brancos e pretos se movem em direções opostas no tabuleiro.

- **Lógica da Função**

Movimento para Frente:

Se o caminho à frente do peão (uma casa para frente) não está ocupado por outra peça (nem branca nem preta), ele pode se mover para essa casa.

Se o peão está na sua posição inicial, ele pode se mover duas casas para frente, desde que ambas as casas estejam desocupadas.

Captura em Diagonal:

Se houver uma peça preta na diagonal à frente do peão (uma casa para frente e uma para a esquerda ou direita), o peão pode se mover para essa casa, capturando a peça preta.

- **Retorno da Função**

A função retorna **lista_movimentos**, que é a lista de todas as possíveis novas posições para o peão, incluindo movimentos normais e capturas.

```

def checar_torre(posicao, cor):
    lista_movimentos = []
    if cor == 'branca':
        # Torres brancas

        lista_aliados = brancas_coord
        lista_inimigos = pretas_coord
    else:
        #Torres pretas

        lista_aliados = pretas_coord
        lista_inimigos = brancas_coord

    for i in range(4): # as 4 direcoes
        caminho = True
        sequencia = 1

        # baixo
        if i == 0:
            x = 0
            y = 1

        # cima
        elif i == 1:
            x = 0
            y = -1

        # direita
        elif i == 2:
            x = 1
            y = 0

        # esquerda
        else:
            x = -1
            y = 0
        while caminho:
            if (posicao[0] + (sequencia * x), posicao[1] + (sequencia * y)) not in lista_aliados and \
            0 <= posicao[0] + (sequencia * x) <= 7 and 0 <= posicao[1] + (sequencia * y) <= 7:
                lista_movimentos.append((posicao[0] + (sequencia * x), posicao[1] + (sequencia * y)))

```

```

        if (posicao[0] + (sequencia * x), posicao[1] + (sequencia * y)) in lista_inimigos:
            caminho = False
            sequencia += 1
        else:
            caminho = True

    return lista_movimentos

```

checar_torre - Usada para calcular todos os movimentos possíveis para a torre. Os parâmetros são os mesmos para todas as funções “checar_peça”, e como já foram explicados em checar_peao, continuarei com a lógica:

- **Lógica da Função**

- **Determinação de Aliados e Inimigos:**

Criamos duas listas para cada jogador armazenando as peças aliadas e inimigas (**lista_aliados** e **lista_inimigos**).

- **Movimento em Quatro Direções:**

A torre pode se mover horizontalmente ou verticalmente qualquer número de casas, mas não pode passar por outras peças;

O loop **for i in range(4)** considera cada uma das quatro direções possíveis.

Dentro de cada direção, um loop **while** percorre o tabuleiro até encontrar uma peça aliada (bloqueando a passagem) ou uma peça inimiga (onde a torre pode capturar e consequentemente parar).

- **Calculando Movimentos Válidos:**

Para cada direção o código verifica se a próxima casa naquela direção está livre (não ocupada por uma peça aliada) e dentro dos limites do tabuleiro (coordenadas entre 0 e 7);

Se a casa está livre, a posição é adicionada à lista de movimentos válidos.

Se uma peça inimiga é encontrada, essa posição também é adicionada (a torre pode capturar a peça inimiga), mas o loop para esta direção termina (a torre não pode passar por uma peça que captura).

- **Retorno da Função:** novamente o retorno é **lista_movimentos**.

```

def checar_cavalo(posicao, cor):
    lista_movimentos = []

    if cor == 'branca':
        # Cavalos brancos
        lista_aliados = brancas_coord
    else:
        #Cavalos pretos
        lista_aliados = pretas_coord

    alvos = [(1, 2), (2, 1), (2, -1), (1, -2), (-1, 2), (-2, 1), (-2, -1), (-1, -2)]
    for i in range(8):
        alvo = (posicao[0] + alvos[i][0], posicao[1] + alvos[i][1])
        if alvo not in lista_aliados and 0 <= alvo[0] <= 7 and 0 <= alvo[1] <= 7:
            lista_movimentos.append(alvo)

    return lista_movimentos

```

checar_cavalo - Responsável pelos cálculos da movimentação do cavalo. Esta função é mais simples pelo fato de que o cavalo pode pular peças:

- **Primeiramente determinamos os aliados.** Neste caso não foi necessário definir os inimigos, pois o seu movimento não é determinado pelas peças inimigas, já que o cavalo se move para 8 posições diferentes possíveis e não é parado por uma peça inimiga ao colidir (neste caso ele captura a peça inimiga). Somente peças aliadas e o limite do tabuleiro tem influências sobre sua movimentação.

- **Movimentos Possíveis do Cavalo:**

O cavalo se move em um padrão de “L”, duas casas em uma direção e uma casa em uma direção perpendicular;

A lista **alvos** contém todos os movimentos possíveis nesse padrão, representando os deslocamentos relativos à posição atual do cavalo.

- **Verificação de Movimentos Válidos:**

O loop `for i in range(8)` itera sobre cada um dos movimentos possíveis em **alvos**;

Para cada movimento possível, a função calcula a nova posição (**alvo**), somando o deslocamento ao **posicao** atual do cavalo.

A função então verifica se essa nova posição **alvo** está dentro dos limites do tabuleiro (**0 a 7 em ambas as direções**) e se não está ocupada por uma peça aliada;

Se ambas as condições forem atendidas, a posição alvo é adicionada à lista `lista_movimentos`.

- **Retorno da Função:** novamente retornamos `lista_movimentos`.

```
def checar_bispo(posicao, cor):
    lista_movimentos = []

    if cor == 'branca':
        # Bispos brancos

        lista_aliados = brancas_coord
        lista_inimigos = pretas_coord
    else:
        # Bispos pretos

        lista_aliados = pretas_coord
        lista_inimigos = brancas_coord

    for i in range(4): # as 4 diagonais
        caminho = True
        sequencia = 1

        # nordeste
        if i == 0:
            x = 1
            y = -1

        # noroeste
        elif i == 1:
            x = -1
            y = -1

        # sudeste
        elif i == 2:
            x = 1
            y = 1

        # sudoeste
        else:
            x = -1
            y = 1

        while caminho:
            if (posicao[0] + (sequencia * x), posicao[1] + (sequencia * y)) not in lista_aliados and \
                0 <= posicao[0] + (sequencia * x) <= 7 and 0 <= posicao[1] + (sequencia * y) <= 7:
                lista_movimentos.append((posicao[0] + (sequencia * x), posicao[1] + (sequencia * y)))
            if (posicao[0] + (sequencia * x), posicao[1] + (sequencia * y)) in lista_inimigos:
                caminho = False
            sequencia += 1
        else:
            caminho = False

    return lista_movimentos
```

checar_bispo - Função responsável pelo cálculo de movimentos possíveis do bispo. Esta função segue uma lógica extremamente similar a da **torre**, a única diferença sendo que o bispo se move na **diagonal** (tendo alterações no o x e o y em conjunto). Como o restante da função é igual, passaremos para a próxima.

```
def checar_rainha(posicao, cor):
    lista_movimentos = []

    # Adicionando os movimentos de torre a rainha
    lista_movimentos_torre = checar_torre(posicao,cor)

    for i in range(len(lista_movimentos_torre)):
        lista_movimentos.append(lista_movimentos_torre[i])

    # Adicionando os movimentos de bispo a rainha
    lista_movimentos_bispo = checar_bispo(posicao, cor)

    for i in range(len(lista_movimentos_bispo)):
```

```

        lista_movimentos.append(lista_movimentos_bispo[i])

    return lista_movimentos

```

checkar_rainha - Calcula todos os movimentos possíveis da rainha. Como a rainha se movimenta de forma que combina os movimentos de uma torre e de um bispo, podendo se mover tanto em linhas retas (horizontalmente, verticalmente) quanto em diagonais, utilizamos de **checkar_torre** e **checkar_bispo**. Adicionamos todos os movimentos válidos da torre e do bispo na lista **lista_movimentos** e a retornamos.

```

def checkar_rei(posicao, cor):
    lista_movimentos = []
    if cor == 'branca':
        # Torres brancas

        lista_aliados = brancas_coord
    else:
        #Torres pretas

        lista_aliados = pretas_coord

    alvos = [(1, 0), (1, 1), (1, -1), (0, 1), (0, -1), (-1, 1), (-1, 0), (-1, -1)]
    for i in range(8):
        alvo = (posicao[0] + alvos[i][0], posicao[1] + alvos[i][1])
        if alvo not in lista_aliados and 0 <= alvo[0] <= 7 and 0 <= alvo[1] <= 7:
            lista_movimentos.append(alvo)

```

checkar_rei - Função que calcula todos os movimentos possíveis do rei. Utilizamos da mesma lógica que **checkar_cavalo**, alterando somente as casas na lista **alvos**, já que o rei só pode se movimentar nas casas adjacentes a si mesmo.

```

def xeque_draw():
    if turno < 2:
        if 'rei' in brancas:
            rei_index = brancas.index('rei')
            rei_loc = brancas_coord[rei_index]
            for i in range(len(opcoes_pretas)):
                if rei_loc in opcoes_pretas[i]:
                    if frame < 30:
                        pygame.draw.rect(tela, 'dark red', [rei_loc[0] * 80 + 1, rei_loc[1] * 80 + 1, 80, 80], 5)

                        espelhado_x = 7 - rei_loc[0]
                        espelhado_y = 7 - rei_loc[1]

                        pygame.draw.rect(tela, 'dark red', [espelhado_x * 80 + 1 + 840, espelhado_y * 80 + 1, 80, 80], 5)

        else:
            if 'rei' in pretas:
                rei_index = pretas.index('rei')
                rei_loc = pretas_coord[rei_index]
                for i in range(len(opcoes_brancas)):
                    if rei_loc in opcoes_brancas[i]:
                        if frame < 30:
                            pygame.draw.rect(tela, 'dark red', [rei_loc[0] * 80 + 1, rei_loc[1] * 80 + 1, 80, 80], 5)

                            espelhado_x = 7 - rei_loc[0]
                            espelhado_y = 7 - rei_loc[1]

                            pygame.draw.rect(tela, 'dark red', [espelhado_x * 80 + 1 + 840, espelhado_y * 80 + 1, 80, 80], 5)

```

xeque_draw - Esta função é responsável pela animação do quadrado do rei piscando em uma situação de xeque:

- **Lógica da Função**

Checamos se a posição do rei está em qualquer uma das listas de movimentos possíveis do adversário, isso indicaria uma situação de xeque;

Se o rei estiver em xeque utilizamos da função **pygame.draw.rect** para desenhar o *outline* do quadrado vermelho escuro na posição onde o rei está.

Além disso, criamos o efeito de piscar utilizando da variável **frame** (você verá em breve, que no loop principal do jogo, ela é incrementada de 0 a 60), somente desenhando o *outline* do quadrado acima se o valor de **frame** for menor do que 30.

Utilizamos da mesma lógica de “espelhamento” utilizada em funções de desenho anteriormente (como o desenho de peças) para garantir que esta animação esteja presente nos dois tabuleiros.

```

def checar_movimentos_validos():
    if turno < 2:
        lista_opcoes = opcoes_branca
    else:
        lista_opcoes = opcoes_pretas
    opcoes_validas = lista_opcoes[selecao]
    return opcoes_validas

```

checar_movimentos_validos - Projetada pra determinar os movimentos válidos para uma peça específica selecionada. Esta função é importante principalmente para os trechos de código que virão adiante.

```

def validos_draw(movimentos):
    for i in range(len(movimentos)):
        pygame.draw.circle(tela, 'red', (movimentos[i][0] * 80 + 40, movimentos[i][1] * 80 + 40), 10)

        x_espelhado = 7 - movimentos[i][0]
        y_espelhado = 7 - movimentos[i][1]

        pygame.draw.circle(tela, 'red', (x_espelhado * 80 + 40 + 840, y_espelhado * 80 + 40), 10)

```

validos_draw - Esta função é usada para visualizar os movimentos válidos de uma peça de xadrez no tabuleiro. Ela desenha círculos vermelhos nas posições para onde a peça selecionada pode legalmente se mover. No loop principal do jogo, passaremos uma variável **movimentos_validos** que recebe os valores da função **checar_movimentos_validos()**. Veremos isto melhor em breve.

```

def banner_fim_draw():
    if vencedor != '':
        pygame.draw.rect(tela, 'black', [0, 270, 1480, 200])
        pygame.draw.rect(tela, 'white', [0, 270, 1480, 200], 4)
        tela.blit(fonte_fim.render(f'{vencedor} VENCEU!', True, 'white'), (410, 300))
        tela.blit(fonte_fim_2.render('Aperte ENTER para jogar novamente', True, 'white'), (500, 400))

```

banner_fim_draw - Esta função é projetada para exibir um *banner* com uma mensagem quando o jogo termina e temos um vencedor.

A função primeiro verifica se a variável **vencedor** é diferente de uma string vazia (""). Se vencedor contém um valor (como 'BRANCO' ou 'PRETO'), isso indica que o jogo terminou com um vencedor.

Feito isso desenhamos um retângulo preto com uma borda branca como *banner* e é mostrada uma mensagem com **BRANCO/PRETO VENCEU!**, dependendo de quem venceu.

Abaixo exibimos uma mensagem de que para jogar novamente, o usuário apertaria **ENTER**. Explicaremos a lógica por trás deste **reset** em breve.

```

# Loop do jogo
opcoes_branca = checar_opcoes(brancas, brancas_coord, 'branca')
opcoes_pretas = checar_opcoes(pretas, pretas_coord, 'preta')
rodando = True
while rodando:
    relogio.tick(fps)
    if frame < 60:
        frame += 1
    else:
        frame = 0
    tela.fill(cor_quadrados_1)
    tabuleiros_draw(tela, (cor_quadrados_1, cor_quadrados_2))
    pecas_draw_um()
    pecas_draw_dois()
    texto_draw()
    capturadas_draw()
    xeque_draw()
    if selecao != 10000:
        movimentos_validos = checar_movimentos_validos()
        validos_draw(movimentos_validos)

```

Chegamos ao **loop principal do jogo**:

Configuração Inicial

- **opcoes_branca**s e **opcoes_preta**s - Calcula os movimentos possíveis para todas as peças brancas e pretas no início do jogo.
- **rodando** - Uma variável booleana usada para manter o loop do jogo em execução. Enquanto **rodando** for **True**, o jogo continua.

Dentro do Loop

- **Controle de Tempo**:
 - **relogio.tick(fps)** - Mantém o jogo rodando à taxa de quadros por segundo definida anteriormente (60).
 - **frame** - Usado para controlar a animação do piscar do xeque.
- **Desenho do Tabuleiro e Peças**:
 - Preenchemos a tela com o cor de fundo **cor_quadrados_1** utilizando da função **tela.fill()**;
 - Desenhamos os dois tabuleiros, utilizando as cores **cor_quadrados_1** e **cor_quadrados_2**;
 - Desenhamos as peças nos tabuleiros utilizando de **pecas_draw_um()** e **pecas_draw_dois()**;
- **Atualizações Visuais e de Estado**:
 - Exibimos o texto indicando de quem é a vez, utilizando de **texto_draw()**;
 - Mostramos as peças capturadas utilizando de **capturadas_draw()**;
 - Destacamos o rei se ele estiver em xeque utilizando de **xeque_draw()**.
- **Movimentos Válidos**:
 - Se uma peça foi selecionada (**selecao != 10000**), calcula e exibe os movimentos válidos para essa peça.

```
# eventos
for evento in pygame.event.get():
    if evento.type == pygame.QUIT:
        rodando = False
    if evento.type == pygame.MOUSEBUTTONDOWN and evento.button == 1 and not fim_de_jogo:
        x_coord = evento.pos[0] // 80
        y_coord = evento.pos[1] // 80

        if evento.pos[0] > 840:
            x_coord = (evento.pos[0] - 840) // 80
            y_coord = evento.pos[1] // 80
            x_coord = 7 - x_coord
            y_coord = 7 - y_coord

        click_coords = (x_coord, y_coord)
        if turno <= 1:
            if click_coords in brancas_coord:
                selecao = brancas_coord.index(click_coords)
                if turno == 0:
                    turno = 1

            if click_coords in movimentos_validos and selecao != 10000:
                brancas_coord[selecao] = click_coords
                if click_coords in pretas_coord:
                    peca_preta = pretas_coord.index(click_coords)
                    pretas_capturadas.append(pretas[peca_preta])
                    if pretas[peca_preta] == 'rei':
                        vencedor = 'BRANCO'
                        pretas.pop(peca_preta)
                        pretas_coord.pop(peca_preta)
                    opcoes_pretas = checar_opcoes(pretas, pretas_coord, 'preta')
                    opcoes_branca = checar_opcoes(brancas, brancas_coord, 'branca')
                    turno = 2
                    selecao = 10000
                    movimentos_validos = []

            if turno > 1:
                if click_coords in pretas_coord:
                    selecao = pretas_coord.index(click_coords)
                    if turno == 2:
                        turno = 3

                if click_coords in movimentos_validos and selecao != 10000:
                    pretas_coord[selecao] = click_coords
                    if click_coords in brancas_coord:
                        peca_branca = brancas_coord.index(click_coords)
```

```

        brancas_capturadas.append(brancas[peca_branca])
        if brancas[peca_branca] == 'rei':
            vencedor = 'PRETO'
            brancas.pop(peca_branca)
            brancas_coord.pop(peca_branca)
        opcoes_pretas = checar_opcoes(pretas, pretas_coord, 'preta')
        opcoes_branca = checar_opcoes(brancas, brancas_coord, 'branca')
        turno = 0
        selecao = 10000
        movimentos_validos = []
    if evento.type == pygame.KEYDOWN and fim_de_jogo:
        if evento.key == pygame.K_RETURN:
            fim_de_jogo = False
            vencedor = ''
            brancas = ['torre', 'cavalo', 'bispo', 'rainha', 'rei', 'bispo', 'cavalo', 'torre',
                       'peao', 'peao', 'peao', 'peao', 'peao', 'peao', 'peao']
            brancas_coord = [(0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7),
                             (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]

            pretas = ['torre', 'cavalo', 'bispo', 'rainha', 'rei', 'bispo', 'cavalo', 'torre',
                      'peao', 'peao', 'peao', 'peao', 'peao', 'peao', 'peao']
            pretas_coord = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0),
                            (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
            brancas_capturadas = []
            pretas_capturadas = []

            turno = 0
            selecao = 10000
            movimentos_validos = []

            opcoes_branca = checar_opcoes(brancas, brancas_coord, 'branca')
            opcoes_pretas = checar_opcoes(pretas, pretas_coord, 'preta')

        if vencedor != '':
            fim_de_jogo = True
            banner_fim_draw()

        pygame.display.flip()
    pygame.quit()

```

Processamento de Eventos

- **Eventos de Saída:** Se o evento detectado é do tipo **pygame.QUIT** (por exemplo, fechar a janela do jogo), a variável **rodando** é definida como **False**, o que termina o loop do jogo.
- **Eventos de Mouse:**
 - Quando um botão do mouse é pressionado (**pygame.MOUSEBUTTONDOWN**), o código calcula as coordenadas do clique no tabuleiro.
 - Se o clique for no segundo tabuleiro (coordenadas maiores que 840 pixels), as coordenadas são ajustadas para corresponder ao primeiro tabuleiro, de forma que um clique numa peça preta no tabuleiro do segundo jogador, selecionará a mesma peça no tabuleiro do primeiro jogador.
 - **click_coords** armazena a posição de clique ajustada no tabuleiro.
 - Dependendo do turno, verifica-se se a peça clicada pertence ao jogador da vez. Se sim, a peça é selecionada (**selecao**) e as ações subsequentes são processadas.

Lógica de Movimentação e Captura de Peças

- **Movimentação de Peças:**
 - Se a posição clicada está na lista de movimentos válidos e uma peça foi selecionada, a peça é movida para a nova posição.
 - Se a nova posição contém uma peça do oponente, esta é capturada e adicionada à lista correspondente de peças capturadas.
 - Se o rei for capturado, o vencedor é declarado.
 - As opções de movimento são atualizadas após cada movimento.

Reinicialização do Jogo

- **Eventos de Teclado:**
 - Se o jogo terminou (**fim_de_jogo**) e o jogador pressiona a tecla ENTER (**pygame.K_RETURN**), o jogo é reiniciado. Todas as variáveis são redefinidas para seus estados iniciais, incluindo as posições das peças, peças capturadas, turno, seleção e movimentos válidos.

Atualizações de Tela e Fim do Jogo

- **Banner de Fim de Jogo:**
 - Se houver um vencedor, a função **banner_fim_draw()** é chamada para exibir a mensagem de fim de jogo.
- **pygame.display.flip()**: Atualiza a tela inteira com tudo o que foi desenhado durante o loop atual.
- **pygame.quit()**: Encerra o jogo quando o loop é finalizado.

 [Conclusão](#)



Conclusão

Neste projeto de xadrez, aplicamos conceitos que aprendemos nas aulas de Computação Gráfica, especialmente em sistemas de coordenadas e translação de objetos, fundamentais para a lógica de movimentação das peças. Utilizamos modelagem geométrica para desenhar o tabuleiro, suas bordas, e elementos visuais como o banner de fim de jogo. Além disso incorporamos técnicas de animação para destacar situações de xeque. Este projeto fortaleceu nosso conhecimento teórico em Computação Gráfica e nos deu experiência prática em design de jogos digitais.