Desenvolvimento de um Jogo Tetris

Marcos Vinicius Dias Oliveira[1]

[1]Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS)
Feira de Santana – BA – Brasil

marcosvdiaso@gmail.com

1. Introdução

O *Tetris* é um jogo clássico de *puzzle*, em que peças de formatos variados caem gradualmente. Durante o jogo, o jogador deve movê-las e rotacioná-las para encaixá-las corretamente no tabuleiro. O objetivo é completar linhas horizontais, que desaparecem ao serem preenchidas, concedendo pontos ao jogador. A *Blizzard Entertainment*, uma desenvolvedora e editora de jogos americana, abriu recentemente uma seleção para contratar jovens talentos para estágio. O processo de seleção consistiu, basicamente, no desenvolvimento de uma versão do jogo *Tetris*.

Este relatório tem como objetivo descrever o processo de desenvolvimento dessa versão do *Tetris*, que foi projetado inicialmente para incluir sete peças, possibilitando movimentação lateral e rotação, a remoção de linhas, contagem de pontuação e uma condição de fim de jogo. Durante o desenvolvimento, buscou-se atender aos requisitos principais de jogabilidade e responsividade, de forma que o sistema pudesse replicar a experiência clássica do *Tetris*. Além disso, também foram considerados aspectos de modularização para facilitar futuras expansões ou ajustes na mecânica do jogo, caso necessário ou desejado.

2. Metodologia

Nesta seção, serão detalhados os requisitos e funcionalidades do jogo, conforme o que foi solicitado no problema original, novas solicitações, as discussões realizadas durante as sessões tutoriais e com base nos sistemas e ferramentas utilizados na elaboração do projeto. Serão abordadas as necessidades identificadas, as soluções propostas e como essas soluções foram implementadas para atender aos requisitos estabelecidos.

2.1. Ferramentas

O sistema foi desenvolvido utilizando a linguagem de programação *Python* 3.12, com o *Visual Studio Code* 1.94.2 como Ambiente Integrado de Desenvolvimento (*IDE*) e o sistema operacional *Windows* 11.

2.2. Requisitos

O desenvolvimento do jogo exigiu o cumprimento de alguns requisitos:

- O tabuleiro tem dimensões de 10 colunas (largura) por 20 linhas (altura), e cada posição do tabuleiro deve estar vazia ou ocupada por parte de uma peça.
- Existem 7 tipos de peças, cada uma com diferentes formatos, e cada peça pode aparecer em várias rotações e posições no topo do tabuleiro.

- O jogador pode mover as peças para ambos os lados, além de as rotacionar.
- As peças continuam caindo até colidirem com outra peça ou com o fundo do tabuleiro.
- Quando uma linha é completamente preenchida com blocos, ela é removida do tabuleiro, e todas as linhas acima descem uma posição.
- O jogador ganha pontos para cada linha removida; no caso de múltiplas linhas removidas simultaneamente, o jogador ganha o dobro de pontos.
- O jogo termina quando uma nova peça não pode ser inserida no tabuleiro.
- As principais tarefas do *software* devem ser implementadas com o conceito de modularização.

Todos os requisitos foram totalmente cumpridos, com exceção do tamanho do tabuleiro e da condição de fim de jogo, que sofreram alterações na versão final do jogo, conforme será explicado posteriormente no relatório. Além dos requisitos iniciais, foi solicitado posteriormente um novo recurso no jogo: uma peça bomba.

2.3. Modularização e Bibliotecas

Uma das principais solicitações para o projeto era: as principais tarefas devem ser implementas utilizando o conceito de modularização. Tendo isso em mente, o código possui no total 10 funções, que foram criadas em um arquivo separado chamado *tetris.py* e importadas para o *main.py.*

A modularização é essencial para a organização e manutenção do código, pois ela permite que cada função trate uma tarefa específica, assim deixando o código mais legível e fácil de bugar. Além disso, a modularização também possibilita a reutilização do código, facilitando a adição de novas funcionalidades no futuro.

Foram também utilizadas quatro bibliotecas diferentes no código, foram elas: *random, time, os* e keyboard. A biblioteca *random* foi utilizada para sorteios dentro do código, como por exemplo escolha aleatória da peça ou decidir se a peça irá cair rotacionada ou não. A biblioteca *time* foi utilizada por conta da função sleep, que serviu para simular o tempo de queda da peça. A biblioteca *os* foi utilizada para poder limpar o terminal em algumas situações no código. A biblioteca *keyboard* foi utilizada para capturar as teclas apertadas pelo usuário e atribuir comandos a elas dentro do jogo, vale ressaltar que essa biblioteca precisa de instalação, por meio do comando no terminal *pip install keyboard* e ela tem problemas de compatibilidade com sistemas *Linux*.

2.4. Tabuleiro

Na primeira versão, o tabuleiro era exatamente conforme o problema pedia, com 10 colunas e 20 linhas. Para isso, foi implementada uma matriz com 20 linhas, cada uma contendo 10 quadrados. Segue a demonstração:

Figura 1. Declaração inicial do tabuleiro.

Porém, após várias versões do programa, o tabuleiro sofreu algumas mudanças. Na versão final, ele foi alterado para uma matriz de 12 colunas e 22 linhas, onde as 2 colunas extras foram inseridas em cada extremidade do tabuleiro para representar as bordas. Além disso, uma linha foi adicionada na parte inferior, representando o fundo, e outra linha foi inserida na parte superior, que é importante para a condição de fim de jogo que será explicada posteriormente. Segue imagem demonstrando o tabuleiro final, durante a execução do código:

Figura 2. Versão final do tabuleiro.

Para poder imprimir o tabuleiro, foi criada uma função chamada *printar_tabuleiro()*, que recebe como parâmetro o tabuleiro. Essa função itera por cada linha do tabuleiro, ou seja, por cada lista da matriz, e imprime cada lista individualmente usando o método *join()*. Isso garante que o tabuleiro não seja exibido no formato de lista, mas sim como um texto normal, dando a impressão de ser uma única estrutura.

2.5. Peças e Movimentação

À princípio, eram 7 tipos de peças, cada uma com um formato diferente. Todas as peças foram criadas por meio de matrizes individuais e, em seguida, foram agrupadas em uma lista de peças. Para a escolha da peça aleatória que será impressa, foi utilizada a função *random.choice()*, que seleciona uma peça aleatória da lista e a atribui à variável de peça aleatória, inserindo-a no tabuleiro.

Para determinar a posição inicial no topo do tabuleiro, foi utilizado *random.randint(),* que sorteia um valor entre 1 e o limite que a peça pode ocupar sem ultrapassar o tabuleiro. Esse valor é então atribuído à variável coluna, que define a posição inicial no topo. Quanto à rotação inicial aleatória, foi utilizado um *random,choice()* booleano. Caso o resultado seja *False*, a peça é inserida em sua rotação padrão; se for *True*, um novo sorteio ocorre, desta vez de inteiros, onde se determina se

a peça deve girar 1, 2 ou 3 vezes, alterando assim sua rotação inicial. Segue abaixo uma imagem que demonstra como esses sorteios foram inseridos no código:

```
peca_aleatoria = random.choice(pecas) # É escolhida uma
# Sorteio para decidir se a peça irá aparecer em sua pos
if random.choice([True, False]):
    for x in range(random.randint(1, 3)): # Caso vá ser
        peca_aleatoria = girar_peca(peca_aleatoria)
coluna = random.randint(1, 10 - len(peca_aleatoria[0]))
```

Figura 3. Sorteios para inserção de nova peça.

Após a nova peça ser inserida no tabuleiro entramos na parte de movimentação. Toda a movimentação da peça é guiada por uma função que é fundamental para o comportamento da peça no tabuleiro, a função *colidir()*. Essa função em resumo, itera pelo eixo x e y da peça aleatória escolhida, e verifica individualmente por cada elemento, a colisão. Para a verificação da colisão são feitas 4 checagens:

- Se não está ultrapassando a margem inferior (para colidir com o fundo);
- Se não está ultrapassando a margem direita ou esquerda (para colidir com as laterais do tabuleiro);
- E se o elemento na linha abaixo, na coluna à esquerda ou na coluna à direita é diferente do bloco que representa o tabuleiro.

Se em qualquer uma dessas checagens retornar *True,* então a função retorna *True*, mas em caso contrário, a função irá retornar *False*. Essa função é utilizada dentro do código principal para: verificar a colisão para permitir a rotação, verificar a colisão para mover para a esquerda ou direita e também é chamada constantemente para permitir a queda do bloco, e caso retorne *True*, para permitir o aparecimento de uma nova peça.

Sobre a movimentação lateral, para isso foi utilizado a biblioteca *keyboard* para capturar a tecla que o usuário está apertando, após uma breve leitura da documentação oficial da biblioteca, foi utilizado do método *keyboard.is_pressed()*. Caso seja capturada a tecla ‘a’, a peça irá mover para a esquerda, já caso seja capturada a tecla ‘d’, a peça irá mover para a direita. Quando uma dessas 2 teclas é pressionada, então o código verifica se não haverá colisão para a esquerda ou direita, dependendo do botão pressionado, e caso não haja colisão ele modifica a variável de coluna, que vai alterar onde a peça será printada no próximo ciclo do loop do código.

Com relação a rotação da peça, utiliza-se o mesmo método para capturar a tecla, desta vez monitorando a tecla ‘w’. Se essa tecla for pressionada, o código atribui o valor da função *girar_peça()* a uma variável *placeholder*, que serve apenas para armazenar temporariamente a forma da peça rotacionada. Em seguida, verifica-se se a variável placeholder, ao ser inserida no tabuleiro no lugar da peça atual, resultará em colisão. Caso ocorra uma colisão, a rotação não é permitida. Porém, se não houver colisão, a variável de peça aleatória recebe o valor da variável *placeholder*, substituindo a peça no tabuleiro.

A função *girar_peca()* realiza a rotação da peça de forma simples. Primeiro, a matriz da peça é invertida utilizando do parâmetro [::1], a matriz é invertida para que a rotação seja no sentido horário. Após isso é aplicado o método *zip(*)* na matriz.

Originalmente, foi considerada a biblioteca *NumPy* para realizar a transposição da matriz, mas o fato de o *zip(*)* não exigir bibliotecas externas ou alterações no código, fez com que eu optasse pelo uso dele. Esse método basicamente vai "agrupar as colunas" da lista em tuplas. Então cada uma dessas tuplas é transformada em uma lista e a função retorna como valor todas essas listas resultantes dentro de uma lista, ou seja, a função retorna como valor uma matriz com as listas resultantes, que nada mais é, que a peça rotacionada.

Além da colisão, da movimentação lateral e da rotação, foi adicionado um extra, a possibilidade de acelerar a queda da peça. Caso o usuário pressione a tecla 's', a variável tempo_queda é diminuida pela metade, tempo_queda é a variável que foi definida como valor do *time.sleep()* utilizado no código, logo quando ela é dividida pela metade, o tempo de queda também reduz pela metade. Caso o usuário solte o botão, então o tempo_queda retorna ao normal.

2.6. Bomba

Foi mencionado anteriormente que à principio eram 7 tipos de peças diferentes. Inicialmente sim, eram apenas 7 modelos distintos de peças, porém após um tempo, a *Blizzard Entertainment* entregou uma surpresa bombástica. A empresa solicitou que fosse acrescentado um novo tipo de peça, a bomba. Uma peça 1x1 que, ao aparecer no topo do tabuleiro, quando ela cair e colidir, ela deve destruir todos os blocos que estiverem posicionados ao seu redor.

A implementação da bomba foi bem simples, primeiramente, ela não é incluída no primeiro sorteio de peças, pois não faria sentido sortear uma bomba sem ter nada para destruir. Após o primeiro sorteio é feito um *.append(bomba)* na lista de peças, para que ela passe a poder ser sorteada.

Então, sempre que uma peça colidir, é feito uma verificação se a peça em questão era uma bomba, caso seja, é chamado a função *peca_bomba()*. Essa função itera por todo o tabuleiro, procurando onde há uma bomba, então ele armazena a linha e coluna da bomba, e altera os blocos ao redor por bloco de tabuleiro. Antes de fazer essa alteração ele faz a verificação de margem, para não trocar a borda ou o fundo do tabuleiro por exemplo.

2.7. Remoção de Linhas e Pontuação

Sempre que uma linha é completada, ela deve ser removida do tabuleiro, todas as linhas acima devem descer, e uma pontuação é adicionada ao jogador com base no número de linhas removidas. Para isso, foi criada uma função chamada *linha_completa()*.

Na primeira versão dessa função, havia um problema: ela verificava se havia alguma linha completa e, caso houvesse, apagava a linha e inseria uma nova linha no topo do tabuleiro, o que era a lógica teoricamente correta. No entanto, a função removia cada linha completa individualmente, mesmo quando várias linhas eram completadas simultaneamente, o que causava problemas na atribuição da pontuação.

Assim, surgiu a versão final dessa função. Agora, a função começa inicializando uma lista vazia que serve para armazenar os índices das linhas completas. Em seguida, é iniciado um *loop* que percorre todas as linhas do tabuleiro (exceto a primeira e a última) e, caso alguma linha esteja completa, o índice dessa linha é adicionado à lista. Após

iterar por todas as linhas, o programa entra em outro *loop* que percorre cada elemento da lista de índices. Dentro desse *loop*, cada linha cujo índice está na lista é deletada e uma nova linha vazia é inserida no topo do tabuleiro após cada remoção de linha. Após remover todas as linhas completas, a função retorna o número de elementos na lista de índices, representando o total de linhas removidas.

Dentro do código principal o valor retornado da função é atribuído a uma variável chamada "linhas_removidas". Durante a execução do programa, a cada loop é verificado se o valor dessa variável é maior que 0, se sim é somado a variável de pontuação, os pontos proporcionais a quantidade de linhas removidas. O cálculo da pontuação com base nas linhas removidas é feito utilizando uma fórmula em que o valor base de 100 é multiplicado por 2 elevado ao número de linhas removidas menos 1, com isso a pontuação sendo na prática uma progressão geométrica.

2.8. Fim de Jogo

Na entrega do problema foi solicitado que a condição de fim de jogo seria: não ser possível acrescentar novas peças no tabuleiro. Porém após testar o *Tetris* original, foi possível constatar que, no jogo, a partida acaba quando alguma peça ultrapassa o topo do tabuleiro, independente se há espaço para novas peças ou não. Dessa forma, com o objetivo de ser fiel a referência original, a condição de fim de jogo criada para esse *Tetris* é a mesma; caso qualquer bloco ultrapasse o topo do tabuleiro, *game over.*

Inicialmente foi preciso então definir o que seria ultrapassar o topo do tabuleiro, para isso então foi necessário alterar o tabuleiro, onde foi acrescentado uma nova linha no topo e caso qualquer bloco esteja fixado nessa linha o jogo acaba. Para isso foi criada a função *game_over(),* essa função recebe uma lista com todos os blocos que representam peças no jogo, e então ela verifica por meio do método *any(),* se qualquer elemento da primeira linha é um desses blocos, caso seja, a função retorna como valor *True,* e em caso contrário retorna *False.* Dentro do código principal, após qualquer bloco ser fixado no tabuleiro, a função *game_over()* é verificada.

2.9. Estrutura do código principal

Como foi mencionado anteriormente, o programa está dividido em 2 arquivos diferentes, *tetris.py*, que contém todas as funções e *main.py*, que é o código principal. Dentro do *main.py* há o loop principal do jogo, esse loop continua rodando enquanto a variável "fim" for *False.*

Dentro do loop a ordem de execução é:

1. *apagar_peca();*
2. Verificar se há alguma linha completa e se houver, somar pontuação com base nas linhas completas;
3. Capturar os botões apertados pelo usuário para movimentação;
4. Verificar colisão:
    1. Caso não haja colisão é adicionado mais 1 ao valor da variável linha;
    2. Caso haja colisão então:

1. Primeiramente é verificado se a peça colidida era uma bomba, caso a resposta seja positiva, é chamada a função *peca_bomba();*
2. Após isso é verificado se é fim de jogo, caso seja, a variável fim é trocada para *True*, encerrando o loop principal do jogo
3. Caso não seja fim de jogo, então é gerada uma nova peça aleatória no topo do tabuleiro

5. *print_peca();*
6. Limpar terminal, com o uso da função *os.system('cls')*
7. *printar_tabuleiro();*
8. Printar o número de pontos logo abaixo do tabuleiro;
9. Um *time.sleep()*, para que o código tenha tempo de exibir as ações que acontecem durante sua execução.

É possível perceber que nessa estrutura foram mencionadas duas novas funções: *print_peca()* e *apagar_peca()*. Essas duas funções são fundamentais para a exibição da peça no tabuleiro. Ambas as funções possuem a mesma estrutura: elas iteram pelos eixos x e y da matriz da peça sorteada e verificam se, na posição (x, y) há um caractere que representa um pedaço do bloco (pois nos blocos há também caracteres vazios, para poder gerar a forma correta). Caso a resposta seja verdadeira, na função *print_peca()* o elemento tabuleiro[linha + x][coluna + y] é substituido pela peça[x][y], enquanto na função *apagar_peca* é substituido pelo caractere que representa o tabuleiro. Nessas funções linha e coluna são as duas variáveis que são manipuladas para determinar a posição da peça dentro do código principal. Linha é sempre incremetada em um quando não há colisão abaixo, já coluna é sorteado um valor inicial, que determinará a posição de surgimento da peça, mas ela é alterada quando é apertado 'a' ou 'd', para movimentar a peça lateralmente.

2.10. EXTRA: Menu Inicial

Além do que foi pedido, foi incluído um recurso adicional, um menu inicial. Nesse menu há três opções: Jogar, Regras e Sair. A exibição do menu inicial e das regras foram atribuídos a duas funções: *menu_inicial()* e *print_regras(),* essas funções foram criadas para que não houvesse um excesso de *prints* no *main.*

Além do menu inicial, também foi adicionada a opção de jogar novamente, após o fim de jogo, caso o usuário tenha interesse.

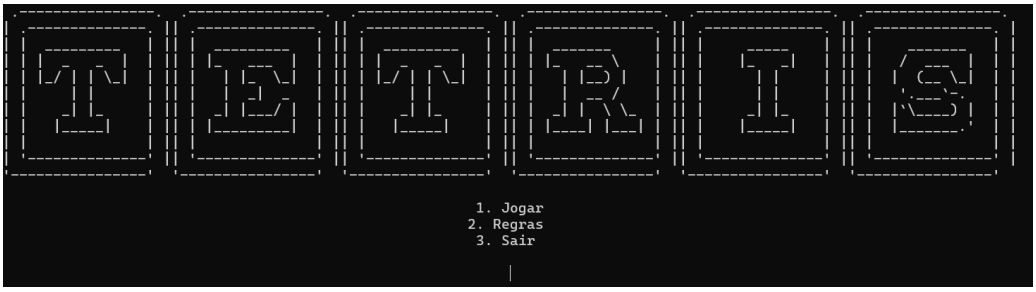

Figura 4. Menu Inicial.

3. Resultados e Discussões

O programa funciona exclusivamente pelo terminal e precisa apenas do teclado. É necessário um interpretador *Python* instalado na máquina para execução do programa. É necessário também a instalação da biblioteca *keyboard,* por meio do comando *pip install keyboard* no terminal.

3.1. Manual de Uso

A fim de facilitar a experiência do usuário, foi criado um manual de uso, que explica os principais pontos do jogo de forma condensada:

- O tabuleiro possui 10 colunas e 21 linhas nas quais as peças conseguem se fixar;
- Caso qualquer peça esteja fixada na primeira linha, o jogo acaba;
- Existem 8 tipos diferentes de peças com diferentes formatos, peças aleatórias aparecem no topo do tabuleiro, em posições e rotações também aleatórias;
- Uma dessas peças é a bomba, quando essa peça encontra colisão, ela destrói todos os blocos ao seu redor;
- Quando uma linha é completamente preenchida com blocos, ela é removida do tabuleiro;
- Ao remover uma linha, todas que estavam acima, descem;
- O jogador ganha pontos por linhas removidas, caso sejam removidas 2, 3 ou 4 linhas simultâneas, a pontuação é dobrada para cada linha;
- Os controles do jogo são: ‘W’ para girar a peça, ‘A’ para mover para a esquerda, ‘D’ para mover para a direita e ‘S’ para acelerar a queda.

4. Conclusão

Em resumo, o programa foi bem-sucedido, todos os requisitos foram atendidos e não foram encontrados *bugs* ou erros que prejudiquem o funcionamento do programa. Todas as funções do jogo funcionam adequadamente, e o *software* foi entregue com recursos extras, como o menu inicial.

Uma possível alteração para o programa, seria a criação de uma função para as movimentações laterais, o que poderia economizar mais linhas no *main*. Além disso outra alteração seria ter feito o fim de jogo como solicitado inicialmente, para que o jogo apenas termine quando não houver mais espaço para nenhuma nova peça. Outra funcionalidade que foi cogitada de ser adicionada, mas acabou não sendo, era armazenar a pontuação em alguma estrutura (lista ou dicionário) após o fim do jogo, assim sendo possível você jogar várias vezes ou diferentes pessoas jogarem, ir armazenando as pontuações e comparar elas depois.

5. Referências Bibliográficas

Documentação da biblioteca *os*, disponível em:
https://docs.python.org/3/library/os.html; Acesso em 20 de Setembro de 2024

W3Schools, *Python Random Module,* disponível em: https://www.w3schools.com/python/module_random.asp; Acesso em 20 de Setembro de 2024

Documentação da biblioteca *keyboard,* disponível em: https://github.com/boppreh/keyboard; Acesso em 27 de Setembro de 2024

Stack Overflow, *"how can i do the tranpose of a matrix in python?",* disponível em: https://stackoverflow.com/questions/73513808/how-can-i-do-the-transpose-of-a-matrix-in-python; Acesso em 04 de Outubro de 2024

Geeks for Geeks, *Python List Slicing,* disponível em: https://www.geeksforgeeks.org/python-list-slicing/; Acesso em 04 de Outubro de 2024

Geeks for Geeks, *Python Reversing List,* disponível em: https://www.geeksforgeeks.org/python-reversing-list/; Acesso em 04 de Outubro de 2024