Aula 6: Movimentação Dinâmica e Posicionamento Tático

Curso: Simulador de Futebol em Python - Parte 2

Professor: Manus AI

Duração: 3 horas

Nível: Avançado

Pré-requisitos: Aulas 1-5 da Parte 1

Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

1. Implementar sistemas de movimentação autônoma para jogadores baseados em suas posições táticas
2. Desenvolver algoritmos de posicionamento dinâmico que respondem ao estado do jogo
3. Criar zonas de influência e sistemas de cobertura defensiva
4. Integrar movimentação individual com estratégias coletivas do time
5. Otimizar algoritmos de pathfinding para simulações em tempo real

Introdução

Na primeira parte do nosso curso, estabelecemos as bases fundamentais do simulador: modelagem de dados, probabilidades, IA básica e interface gráfica. Agora, na Parte 2, vamos elevar o nível de realismo e complexidade, começando com um dos aspectos mais cruciais do futebol moderno: a movimentação inteligente e o posicionamento tático.

No futebol real, cada jogador não apenas reage ao que acontece com a bola, mas também se posiciona estrategicamente no campo, antecipando jogadas, cobrindo espaços e criando oportunidades. Esta movimentação constante e coordenada é o que distingue equipes amadoras de profissionais, e é exatamente isso que vamos implementar em nosso simulador.

A movimentação dinâmica vai muito além de simplesmente "correr atrás da bola". Envolve conceitos complexos como:

- **Posicionamento por função**: Um zagueiro central se comporta diferentemente de um ponta-direita
- **Movimentação sem bola**: Criar espaços, oferecer opções de passe, pressionar

adversários

- **Coordenação coletiva**: Manter linhas defensivas, criar triângulos de passe, executar movimentos ensaiados
- **Adaptação tática**: Responder a mudanças no jogo, como expulsões ou alterações táticas do adversário

1. Fundamentos da Movimentação no Futebol

1.1 Tipos de Movimentação

No futebol, existem diferentes tipos de movimentação que precisamos modelar em nosso simulador:

Movimentação com a Bola:

- Condução simples (correr com a bola)
- Drible (superar adversários)
- Proteção da bola (usar o corpo para proteger)

Movimentação sem a Bola (Ofensiva):

- Buscar espaços livres para receber passes
- Criar linhas de passe para companheiros
- Movimentos de distração para abrir espaços
- Corridas de apoio e sobreposição

Movimentação sem a Bola (Defensiva):

- Marcação individual (acompanhar um adversário específico)
- Marcação por zona (cobrir uma área do campo)
- Pressão coordenada (forçar erros do adversário)
- Cobertura defensiva (apoiar companheiros)

1.2 Algoritmos de Pathfinding para Futebol

Diferentemente de jogos de estratégia onde unidades se movem em grids, no futebol o movimento é contínuo e deve considerar múltiplos fatores simultaneamente. Vamos implementar um sistema híbrido que combina:

*A Modificado para Futebol:**

Python

```python
import heapq
import math
from typing import List, Tuple, Optional

class PosicaoTatica:
    def __init__(self, x: float, y: float, prioridade: int = 0):
        self.x = x
        self.y = y
        self.prioridade = prioridade  # 0 = baixa, 10 = alta
        self.ocupada = False
        self.jogador_responsavel = None

class MotorMovimentacao:
    def __init__(self, largura_campo: float = 100.0, altura_campo: float =
60.0):
        self.largura_campo = largura_campo
        self.altura_campo = altura_campo
        self.posicoes_taticas = {}  # Dicionário por formação
        self._inicializar_posicoes_taticas()

    def calcular_caminho_otimo(self, origem: Tuple[float, float],
                               destino: Tuple[float, float],
                               obstaculos: List[Tuple[float, float]]) ->
List[Tuple[float, float]]:
        """
        Implementa A* modificado para movimentação no futebol.
        Considera não apenas a distância, mas também:
        - Proximidade de adversários (obstáculos dinâmicos)
        - Zonas de perigo (área adversária para defensores)
        - Preferências táticas (laterais preferem as bordas)
        """

        def heuristica(pos1: Tuple[float, float], pos2: Tuple[float, float])
-> float:
            return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)

        def custo_movimento(pos_atual: Tuple[float, float],
                            pos_proxima: Tuple[float, float],
                            obstaculos: List[Tuple[float, float]]) -> float:
            custo_base = heuristica(pos_atual, pos_proxima)

            # Penalidade por proximidade de adversários
            for obstaculo in obstaculos:
                distancia_obstaculo = heuristica(pos_proxima, obstaculo)
                if distancia_obstaculo < 3.0:  # 3 metros de "zona de perigo"
                    custo_base += (3.0 - distancia_obstaculo) * 2.0
```

```
            return custo_base

        # Implementação simplificada do A*
        fila_prioridade = [(0, origem)]
        custos = {origem: 0}
        pais = {origem: None}

        while fila_prioridade:
            custo_atual, pos_atual = heapq.heappop(fila_prioridade)

            if self._posicoes_proximas(pos_atual, destino, tolerancia=1.0):
                # Reconstruir caminho
                caminho = []
                while pos_atual is not None:
                    caminho.append(pos_atual)
                    pos_atual = pais[pos_atual]
                return caminho[::-1]

            # Explorar vizinhos (8 direções + algumas diagonais)
            for dx, dy in [(-1,0), (1,0), (0,-1), (0,1), (-1,-1), (-1,1),
(1,-1), (1,1)]:
                nova_pos = (pos_atual[0] + dx, pos_atual[1] + dy)

                # Verificar limites do campo
                if not self._posicao_valida(nova_pos):
                    continue

                novo_custo = custos[pos_atual] + custo_movimento(pos_atual,
nova_pos, obstaculos)

                if nova_pos not in custos or novo_custo < custos[nova_pos]:
                    custos[nova_pos] = novo_custo
                    prioridade = novo_custo + heuristica(nova_pos, destino)
                    heapq.heappush(fila_prioridade, (prioridade, nova_pos))
                    pais[nova_pos] = pos_atual

        return [origem, destino]  # Caminho direto se A* falhar
```

1.3 Sistema de Coordenadas e Mapeamento

Para integrar a movimentação com nossa interface Pygame, precisamos de um sistema robusto de conversão entre coordenadas do mundo do jogo e coordenadas da tela:

Python

```
class SistemaCoordenadas:
    def __init__(self, largura_campo_real: float, altura_campo_real: float,
```

```
                 largura_tela: int, altura_tela: int, margem: int = 50):
        self.largura_campo_real = largura_campo_real
        self.altura_campo_real = altura_campo_real
        self.largura_tela = largura_tela
        self.altura_tela = altura_tela
        self.margem = margem

        # Calcular fatores de escala
        self.escala_x = (largura_tela - 2 * margem) / largura_campo_real
        self.escala_y = (altura_tela - 2 * margem) / altura_campo_real

        # Usar a menor escala para manter proporções
        self.escala = min(self.escala_x, self.escala_y)

        # Calcular offsets para centralizar o campo
        self.offset_x = (largura_tela - largura_campo_real * self.escala) / 2
        self.offset_y = (altura_tela - altura_campo_real * self.escala) / 2

    def mundo_para_tela(self, x_mundo: float, y_mundo: float) -> Tuple[int,
int]:
        """Converte coordenadas do mundo do jogo para coordenadas da tela."""
        x_tela = int(x_mundo * self.escala + self.offset_x)
        y_tela = int(y_mundo * self.escala + self.offset_y)
        return (x_tela, y_tela)

    def tela_para_mundo(self, x_tela: int, y_tela: int) -> Tuple[float,
float]:
        """Converte coordenadas da tela para coordenadas do mundo do jogo."""
        x_mundo = (x_tela - self.offset_x) / self.escala
        y_mundo = (y_tela - self.offset_y) / self.escala
        return (x_mundo, y_mundo)
```

2. Posicionamento Tático por Função

2.1 Mapeamento de Posições Base

Cada formação tática define posições base para os jogadores. Vamos expandir nossa classe `Time` para incluir um sistema mais sofisticado de posicionamento:

Python

```
class FormacaoTatica:
    def __init__(self, nome: str, posicoes_base: dict):
        self.nome = nome
        self.posicoes_base = posicoes_base  # {posicao: [(x, y), ...]}
        self.instrucoes_especiais = {}
```

```
    def obter_posicao_base(self, posicao: Posicao, indice: int = 0) ->
Tuple[float, float]:
        """Retorna a posição base para um jogador de determinada função."""
        if posicao in self.posicoes_base:
            posicoes = self.posicoes_base[posicao]
            if indice < len(posicoes):
                return posicoes[indice]
        return (50.0, 30.0)  # Posição padrão no meio do campo

# Exemplo de formação 4-4-2
FORMACAO_442 = FormacaoTatica("4-4-2", {
    Posicao.GOLEIRO: [(5.0, 30.0)],
    Posicao.ZAGUEIRO: [(15.0, 15.0), (15.0, 45.0), (20.0, 20.0), (20.0,
40.0)],
    Posicao.MEIA: [(35.0, 10.0), (35.0, 50.0), (45.0, 20.0), (45.0, 40.0)],
    Posicao.ATACANTE: [(75.0, 25.0), (75.0, 35.0)]
})
```

2.2 Movimentação Baseada em Função

Cada posição tem comportamentos específicos de movimentação que devem ser implementados:

Python

```
class ComportamentoMovimentacao:
    @staticmethod
    def calcular_movimento_zagueiro(jogador: Jogador, estado_partida:
EstadoPartida) -> Tuple[float, float]:
        """
        Zagueiros priorizam:
        1. Manter linha defensiva
        2. Cobrir espaços perigosos
        3. Acompanhar atacantes adversários
        """
        posicao_base = jogador.posicao_tatica_base

        # Se o time adversário tem a bola, ajustar posição defensivamente
        if estado_partida.time_com_bola != jogador.time:
            # Encontrar atacante adversário mais próximo
            atacante_mais_proximo =
ComportamentoMovimentacao._encontrar_atacante_mais_proximo(
                jogador, estado_partida
            )

            if atacante_mais_proximo:
```

```
                # Posicionar-se entre o atacante e o gol
                x_gol = 5.0 if jogador.time == estado_partida.time_casa else
95.0
                x_atacante = atacante_mais_proximo.posicao_atual[0]
                y_atacante = atacante_mais_proximo.posicao_atual[1]

                # Calcular posição de interceptação
                x_ideal = (x_gol + x_atacante) / 2
                y_ideal = y_atacante

                # Mesclar com posição base para não sair muito da formação
                x_final = (posicao_base[0] * 0.3) + (x_ideal * 0.7)
                y_final = (posicao_base[1] * 0.3) + (y_ideal * 0.7)

                return (x_final, y_final)

        # Se o próprio time tem a bola, manter posição base com pequenos
ajustes
        return posicao_base

    @staticmethod
    def calcular_movimento_meia(jogador: Jogador, estado_partida:
EstadoPartida) -> Tuple[float, float]:
        """
        Meio-campistas são os mais dinâmicos:
        1. Oferecer opções de passe quando o time tem a bola
        2. Pressionar o portador da bola quando o adversário tem posse
        3. Manter equilíbrio entre ataque e defesa
        """
        posicao_base = jogador.posicao_tatica_base

        if estado_partida.time_com_bola == jogador.time:
            # Time tem a bola - buscar espaços para receber passes
            jogador_com_bola = estado_partida.jogador_com_bola

            if jogador_com_bola != jogador:
                # Calcular posição ideal para receber passe
                x_bola, y_bola = jogador_com_bola.posicao_atual

                # Buscar espaço livre próximo à bola, mas não muito próximo
                distancia_ideal = 15.0  # 15 metros da bola
                angulo = math.atan2(posicao_base[1] - y_bola,
posicao_base[0] - x_bola)

                x_ideal = x_bola + distancia_ideal * math.cos(angulo)
                y_ideal = y_bola + distancia_ideal * math.sin(angulo)

                # Verificar se a posição está livre de adversários
```

```
                if ComportamentoMovimentacao._posicao_livre_de_adversarios(
                    (x_ideal, y_ideal), estado_partida, raio_seguranca=5.0
                ):
                    return (x_ideal, y_ideal)

        else:
            # Adversário tem a bola - pressionar ou cobrir
            jogador_com_bola = estado_partida.jogador_com_bola
            x_bola, y_bola = jogador_com_bola.posicao_atual

            # Calcular posição de pressão
            distancia_pressao = 3.0  # 3 metros do portador da bola
            angulo_pressao = math.atan2(y_bola - posicao_base[1], x_bola -
posicao_base[0])

            x_pressao = x_bola - distancia_pressao * math.cos(angulo_pressao)
            y_pressao = y_bola - distancia_pressao * math.sin(angulo_pressao)

            return (x_pressao, y_pressao)

        return posicao_base

    @staticmethod
    def calcular_movimento_atacante(jogador: Jogador, estado_partida:
EstadoPartida) -> Tuple[float, float]:
        """
        Atacantes focam em:
        1. Buscar posições de finalização
        2. Criar espaços para companheiros
        3. Pressionar defensores adversários
        """
        posicao_base = jogador.posicao_tatica_base

        if estado_partida.time_com_bola == jogador.time:
            # Buscar posição de ataque
            x_gol_adversario = 95.0 if jogador.time ==
estado_partida.time_casa else 5.0

            # Movimento em direção ao gol, mas considerando impedimento
            linha_impedimento =
ComportamentoMovimentacao._calcular_linha_impedimento(
                jogador, estado_partida
            )

            x_ideal = min(x_gol_adversario - 5.0, linha_impedimento - 1.0)
            y_ideal = posicao_base[1] + random.uniform(-10.0, 10.0)  #
Variação lateral
```

```
            return (x_ideal, max(5.0, min(55.0, y_ideal)))

        else:
            # Pressionar defensores adversários
            zagueiro_mais_proximo =
ComportamentoMovimentacao._encontrar_zagueiro_mais_proximo(
                jogador, estado_partida
            )

            if zagueiro_mais_proximo:
                x_zag, y_zag = zagueiro_mais_proximo.posicao_atual
                # Posicionar-se próximo para pressionar
                return (x_zag + 2.0, y_zag)

        return posicao_base
```

2.3 Sistema de Zonas de Influência

Cada jogador possui uma "zona de influência" - uma área do campo onde ele é mais efetivo. Isso ajuda a coordenar movimentos e evitar que jogadores se aglomerem:

Python

```
class ZonaInfluencia:
    def __init__(self, centro: Tuple[float, float], raio: float,
intensidade: float = 1.0):
        self.centro = centro
        self.raio = raio
        self.intensidade = intensidade  # 0.0 a 1.0

    def calcular_influencia(self, posicao: Tuple[float, float]) -> float:
        """Calcula a influência desta zona em uma posição específica."""
        distancia = math.sqrt(
            (posicao[0] - self.centro[0])**2 +
            (posicao[1] - self.centro[1])**2
        )

        if distancia <= self.raio:
            # Influência decresce linearmente com a distância
            influencia_normalizada = 1.0 - (distancia / self.raio)
            return influencia_normalizada * self.intensidade

        return 0.0

class GerenciadorZonas:
    def __init__(self):
        self.zonas_por_jogador = {}
```

```
    def atualizar_zona_jogador(self, jogador: Jogador):
        """Atualiza a zona de influência de um jogador baseada em sua
posição e função."""
        raio_base = {
            Posicao.GOLEIRO: 15.0,
            Posicao.ZAGUEIRO: 12.0,
            Posicao.MEIA: 10.0,
            Posicao.ATACANTE: 8.0
        }

        raio = raio_base.get(jogador.posicao, 10.0)
        intensidade = jogador.posicionamento / 100.0  # Atributo do jogador

        self.zonas_por_jogador[jogador.id] = ZonaInfluencia(
            jogador.posicao_atual, raio, intensidade
        )

    def calcular_conflito_zonas(self, posicao: Tuple[float, float],
                                time: Time) -> float:
        """
        Calcula o nível de conflito (sobreposição) de zonas em uma posição.
        Retorna valor entre 0.0 (sem conflito) e 1.0 (conflito máximo).
        """
        influencia_total = 0.0

        for jogador in time.jogadores:
            if jogador.id in self.zonas_por_jogador:
                zona = self.zonas_por_jogador[jogador.id]
                influencia_total += zona.calcular_influencia(posicao)

        return min(1.0, influencia_total)
```

3. Movimentação Coordenada e Formações Dinâmicas

3.1 Manutenção de Linhas Táticas

Uma das características mais importantes do futebol moderno é a manutenção de linhas táticas coordenadas. Vamos implementar um sistema que garante que os jogadores mantenham suas posições relativas:

Python

```
class CoordenadorLinhas:
    def __init__(self):
        self.linhas_defensivas = {}
```

```
        self.linhas_ofensivas = {}

    def calcular_linha_defensiva(self, time: Time, estado_partida:
EstadoPartida) -> float:
        """
        Calcula a posição X da linha defensiva baseada na posição dos
zagueiros.
        """
        zagueiros = [j for j in time.jogadores if j.posicao ==
Posicao.ZAGUEIRO]

        if not zagueiros:
            return 20.0  # Posição padrão

        # A linha defensiva é determinada pelo zagueiro mais avançado
        x_mais_avancado = max(z.posicao_atual[0] for z in zagueiros)

        # Ajustar baseado na posse de bola
        if estado_partida.time_com_bola == time:
            # Time tem a bola - linha pode subir
            x_mais_avancado = min(x_mais_avancado + 5.0, 45.0)
        else:
            # Adversário tem a bola - linha recua
            x_mais_avancado = max(x_mais_avancado - 3.0, 15.0)

        return x_mais_avancado

    def ajustar_posicoes_para_linha(self, time: Time, linha_defensiva:
float):
        """
        Ajusta as posições dos jogadores para manter a linha defensiva.
        """
        for jogador in time.jogadores:
            if jogador.posicao == Posicao.ZAGUEIRO:
                x_atual, y_atual = jogador.posicao_atual
                # Ajustar X para manter a linha, preservar Y
                jogador.posicao_objetivo = (linha_defensiva, y_atual)

            elif jogador.posicao == Posicao.MEIA:
                # Meio-campistas ficam 10-15 metros à frente da linha
defensiva
                x_atual, y_atual = jogador.posicao_atual
                x_ideal = linha_defensiva + 12.0
                jogador.posicao_objetivo = (x_ideal, y_atual)
```

3.2 Movimentos Coletivos Coordenados

Vamos implementar alguns movimentos coletivos clássicos do futebol:

Python

```
class MovimentosColetivos:
    @staticmethod
    def executar_pressing_coordenado(time: Time, estado_partida:
EstadoPartida):
        """
        Implementa pressing coordenado - todos os jogadores próximos
pressionam simultaneamente.
        """
        jogador_com_bola = estado_partida.jogador_com_bola

        if not jogador_com_bola or jogador_com_bola.time == time:
            return  # Não pressionar se o próprio time tem a bola

        x_bola, y_bola = jogador_com_bola.posicao_atual

        # Encontrar jogadores próximos à bola (raio de 15 metros)
        jogadores_proximos = []
        for jogador in time.jogadores:
            distancia = math.sqrt(
                (jogador.posicao_atual[0] - x_bola)**2 +
                (jogador.posicao_atual[1] - y_bola)**2
            )
            if distancia <= 15.0:
                jogadores_proximos.append(jogador)

        # Coordenar movimento de pressing
        for jogador in jogadores_proximos:
            # Calcular posição de pressing (próximo à bola, mas não todos no
mesmo lugar)
            angulo_base = math.atan2(y_bola - jogador.posicao_atual[1],
                                     x_bola - jogador.posicao_atual[0])

            # Adicionar pequena variação angular para evitar sobreposição
            variacao_angular = random.uniform(-0.3, 0.3)
            angulo_final = angulo_base + variacao_angular

            distancia_pressing = 2.0  # 2 metros do portador da bola
            x_pressing = x_bola - distancia_pressing * math.cos(angulo_final)
            y_pressing = y_bola - distancia_pressing * math.sin(angulo_final)

            jogador.posicao_objetivo = (x_pressing, y_pressing)
            jogador.modo_movimento = "pressing"

    @staticmethod
```

```
    def executar_triangulacao_passe(jogador_com_bola: Jogador, time: Time):
        """
        Cria triângulos de passe - dois jogadores se posicionam para formar
        opções de passe em ângulos diferentes.
        """
        x_bola, y_bola = jogador_com_bola.posicao_atual

        # Encontrar dois companheiros mais próximos
        companheiros = [j for j in time.jogadores if j != jogador_com_bola]
        companheiros.sort(key=lambda j: math.sqrt(
            (j.posicao_atual[0] - x_bola)**2 + (j.posicao_atual[1] -
y_bola)**2
        ))

        if len(companheiros) >= 2:
            # Primeiro companheiro - posição de apoio próximo
            comp1 = companheiros[0]
            angulo1 = math.pi / 4  # 45 graus
            distancia1 = 8.0
            x1 = x_bola + distancia1 * math.cos(angulo1)
            y1 = y_bola + distancia1 * math.sin(angulo1)
            comp1.posicao_objetivo = (x1, y1)

            # Segundo companheiro - posição de apoio distante
            comp2 = companheiros[1]
            angulo2 = -math.pi / 4  # -45 graus
            distancia2 = 12.0
            x2 = x_bola + distancia2 * math.cos(angulo2)
            y2 = y_bola + distancia2 * math.sin(angulo2)
            comp2.posicao_objetivo = (x2, y2)
```

4. Algoritmos de Movimentação Suave

4.1 Interpolação e Suavização de Movimento

Para que a movimentação pareça natural, não podemos simplesmente "teleportar" jogadores para suas posições objetivo. Precisamos de movimento suave e realista:

Python

```
class MotorMovimentacaoSuave:
    def __init__(self):
        self.velocidade_maxima = {
            Posicao.GOLEIRO: 3.0,      # m/s
            Posicao.ZAGUEIRO: 6.0,     # m/s
            Posicao.MEIA: 7.0,         # m/s
```

```
            Posicao.ATACANTE: 8.0      # m/s
        }

    def atualizar_posicao_jogador(self, jogador: Jogador, delta_tempo:
float):
        """
        Atualiza a posição do jogador movendo-o suavemente em direção ao
objetivo.
        """
        if not hasattr(jogador, 'posicao_objetivo'):
            return

        x_atual, y_atual = jogador.posicao_atual
        x_objetivo, y_objetivo = jogador.posicao_objetivo

        # Calcular vetor de movimento
        dx = x_objetivo - x_atual
        dy = y_objetivo - y_atual
        distancia = math.sqrt(dx**2 + dy**2)

        if distancia < 0.1:  # Já chegou ao destino
            return

        # Normalizar vetor de direção
        dx_norm = dx / distancia
        dy_norm = dy / distancia

        # Aplicar velocidade máxima do jogador
        velocidade_base = self.velocidade_maxima.get(jogador.posicao, 6.0)

        # Ajustar velocidade baseada em atributos do jogador
        velocidade_real = velocidade_base * (jogador.aceleracao / 100.0) *
(jogador.forma_fisica / 100.0)

        # Calcular movimento neste frame
        movimento_x = dx_norm * velocidade_real * delta_tempo
        movimento_y = dy_norm * velocidade_real * delta_tempo

        # Aplicar movimento, mas não ultrapassar o objetivo
        if abs(movimento_x) > abs(dx):
            movimento_x = dx
        if abs(movimento_y) > abs(dy):
            movimento_y = dy

        # Atualizar posição
        nova_x = x_atual + movimento_x
        nova_y = y_atual + movimento_y
```

```
        # Garantir que está dentro dos limites do campo
        nova_x = max(0.0, min(100.0, nova_x))
        nova_y = max(0.0, min(60.0, nova_y))

        jogador.posicao_atual = (nova_x, nova_y)

        # Atualizar fadiga baseada no movimento
        distancia_percorrida = math.sqrt(movimento_x**2 + movimento_y**2)
        fadiga_adicional = distancia_percorrida * 0.1
        jogador.forma_fisica = max(0, jogador.forma_fisica -
fadiga_adicional)
```

4.2 Evitando Colisões e Sobreposições

Jogadores não podem ocupar o mesmo espaço. Vamos implementar um sistema de evitação de colisões:

Python

```
class SistemaColisoes:
    def __init__(self, raio_jogador: float = 0.5):
        self.raio_jogador = raio_jogador

    def verificar_colisao(self, pos1: Tuple[float, float],
                          pos2: Tuple[float, float]) -> bool:
        """Verifica se duas posições colidem."""
        distancia = math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] -
pos2[1])**2)
        return distancia < (self.raio_jogador * 2)

    def resolver_colisoes(self, jogadores: List[Jogador]):
        """
        Resolve colisões entre jogadores ajustando suas posições.
        """
        for i, jogador1 in enumerate(jogadores):
            for j, jogador2 in enumerate(jogadores[i+1:], i+1):
                if self.verificar_colisao(jogador1.posicao_atual,
jogador2.posicao_atual):
                    # Calcular vetor de separação
                    x1, y1 = jogador1.posicao_atual
                    x2, y2 = jogador2.posicao_atual

                    dx = x1 - x2
                    dy = y1 - y2
                    distancia = math.sqrt(dx**2 + dy**2)

                    if distancia == 0:  # Posições idênticas
```

```
                    dx, dy = random.uniform(-1, 1), random.uniform(-1, 1)
                    distancia = math.sqrt(dx**2 + dy**2)

                # Normalizar e aplicar separação mínima
                dx_norm = dx / distancia
                dy_norm = dy / distancia

                separacao_minima = self.raio_jogador * 2.1
                movimento_necessario = (separacao_minima - distancia) / 2

                # Mover ambos os jogadores para resolver a colisão
                jogador1.posicao_atual = (
                    x1 + dx_norm * movimento_necessario,
                    y1 + dy_norm * movimento_necessario
                )
                jogador2.posicao_atual = (
                    x2 - dx_norm * movimento_necessario,
                    y2 - dy_norm * movimento_necessario
                )
```

5. Integração com o Motor de Jogo Existente

5.1 Atualizando o Loop Principal

Agora precisamos integrar nosso sistema de movimentação com o motor de eventos existente:

Python

```python
class MotorEventosJogo:
    def __init__(self, seed=None):
        # ... inicialização anterior ...
        self.motor_movimentacao = MotorMovimentacaoSuave()
        self.coordenador_linhas = CoordenadorLinhas()
        self.gerenciador_zonas = GerenciadorZonas()
        self.sistema_colisoes = SistemaColisoes()

    def simular_tick(self, estado: EstadoPartida):
        """
        Versão atualizada do tick que inclui movimentação avançada.
        """
        # 1. Atualizar objetivos de posicionamento para todos os jogadores
        self._atualizar_objetivos_posicionamento(estado)

        # 2. Atualizar posições físicas dos jogadores
        self._atualizar_movimentacao_fisica(estado)
```

```
        # 3. Resolver colisões
        todos_jogadores = estado.time_casa.jogadores +
estado.time_fora.jogadores
        self.sistema_colisoes.resolver_colisoes(todos_jogadores)

        # 4. Atualizar zonas de influência
        for jogador in todos_jogadores:
            self.gerenciador_zonas.atualizar_zona_jogador(jogador)

        # 5. Processar decisões dos jogadores (lógica anterior)
        for jogador in todos_jogadores:
            acao = self.motor_ia.decidir_acao(jogador, estado,

jogador.time.instrucoes_treinador)
            self._processar_acao_jogador(jogador, acao, estado)

        # 6. Processar eventos gerais
        self._processar_eventos_gerais(estado)

        # 7. Avançar tempo
        estado.tempo_atual += TEMPO_TICK

        return estado

    def _atualizar_objetivos_posicionamento(self, estado: EstadoPartida):
        """Calcula onde cada jogador deveria estar posicionado."""

        # Atualizar linhas táticas
        linha_def_casa = self.coordenador_linhas.calcular_linha_defensiva(
            estado.time_casa, estado
        )
        linha_def_fora = self.coordenador_linhas.calcular_linha_defensiva(
            estado.time_fora, estado
        )

        # Ajustar posições dos jogadores do time da casa
        for jogador in estado.time_casa.jogadores:
            if jogador.posicao == Posicao.ZAGUEIRO:
                posicao_objetivo =
ComportamentoMovimentacao.calcular_movimento_zagueiro(
                    jogador, estado
                )
            elif jogador.posicao == Posicao.MEIA:
                posicao_objetivo =
ComportamentoMovimentacao.calcular_movimento_meia(
                    jogador, estado
                )
```

```
            elif jogador.posicao == Posicao.ATACANTE:
                posicao_objetivo =
ComportamentoMovimentacao.calcular_movimento_atacante(
                    jogador, estado
                )
            else:  # Goleiro
                posicao_objetivo = jogador.posicao_tatica_base

            jogador.posicao_objetivo = posicao_objetivo

        # Repetir para time de fora
        for jogador in estado.time_fora.jogadores:
            # ... lógica similar ...
            pass

    def _atualizar_movimentacao_fisica(self, estado: EstadoPartida):
        """Atualiza as posições físicas dos jogadores."""
        for jogador in estado.time_casa.jogadores +
estado.time_fora.jogadores:
            self.motor_movimentacao.atualizar_posicao_jogador(jogador,
TEMPO_TICK)
```

6. Considerações Especiais por Posição

6.1 Comportamento Específico do Goleiro

O goleiro tem comportamentos únicos que merecem atenção especial:

Python

```
class ComportamentoGoleiro:
    @staticmethod
    def calcular_posicionamento_goleiro(goleiro: Jogador, estado_partida:
EstadoPartida) -> Tuple[float, float]:
        """
        Goleiro se posiciona baseado na posição da bola e ameaças.
        """
        x_bola, y_bola = estado_partida.posicao_bola
        x_gol = 5.0 if goleiro.time == estado_partida.time_casa else 95.0

        # Posição base do goleiro (linha do gol)
        x_goleiro = x_gol + (1.0 if goleiro.time == estado_partida.time_casa
else -1.0)

        # Ajustar Y baseado na posição da bola
        # Goleiro se move lateralmente para cobrir o ângulo de chute
```

```
        if abs(x_bola - x_gol) > 20.0:  # Bola longe do gol
            # Posição central
            y_goleiro = 30.0
        else:
            # Bola próxima - ajustar para cobrir ângulo
            y_goleiro = y_bola + (30.0 - y_bola) * 0.3  # Interpolar com
centro

        # Limitar movimento do goleiro à área
        y_goleiro = max(20.0, min(40.0, y_goleiro))

        return (x_goleiro, y_goleiro)

    @staticmethod
    def decidir_saida_goleiro(goleiro: Jogador, estado_partida:
EstadoPartida) -> bool:
        """
        Decide se o goleiro deve sair do gol para interceptar a bola.
        """
        x_bola, y_bola = estado_partida.posicao_bola
        x_gol = 5.0 if goleiro.time == estado_partida.time_casa else 95.0

        # Condições para saída:
        # 1. Bola próxima ao gol (menos de 15 metros)
        # 2. Adversário com a bola
        # 3. Goleiro tem boa decisão e compostura

        distancia_bola_gol = abs(x_bola - x_gol)

        if (distancia_bola_gol < 15.0 and
            estado_partida.time_com_bola != goleiro.time and
            goleiro.decisao > 70 and
            goleiro.compostura > 60):

            # Probabilidade de saída baseada nos atributos
            prob_saida = (goleiro.decisao + goleiro.compostura) / 200.0
            return random.random() < prob_saida

        return False
```

6.2 Movimentação de Laterais

Os laterais têm um papel único, precisando equilibrar defesa e apoio ao ataque:

Python

```python
class ComportamentoLateral:
    @staticmethod
    def calcular_movimento_lateral(lateral: Jogador, estado_partida:
EstadoPartida) -> Tuple[float, float]:
        """
        Laterais têm comportamento híbrido entre zagueiro e meia.
        """
        posicao_base = lateral.posicao_tatica_base
        x_base, y_base = posicao_base

        if estado_partida.time_com_bola == lateral.time:
            # Time tem a bola - apoiar o ataque

            # Se a bola está do seu lado do campo, subir para apoiar
            x_bola, y_bola = estado_partida.posicao_bola

            # Determinar se é lateral direito ou esquerdo
            eh_lateral_direito = y_base > 30.0

            if ((eh_lateral_direito and y_bola > 40.0) or
                (not eh_lateral_direito and y_bola < 20.0)):
                # Bola do seu lado - subir para apoiar
                x_apoio = min(x_base + 20.0, 80.0)
                y_apoio = y_base
                return (x_apoio, y_apoio)
            else:
                # Bola do outro lado - manter posição mais recuada
                x_cobertura = x_base
                y_cobertura = y_base
                return (x_cobertura, y_cobertura)

        else:
            # Adversário tem a bola - priorizar defesa
            return
ComportamentoMovimentacao.calcular_movimento_zagueiro(lateral,
estado_partida)
```

7. Implementação Prática: Integrando ao Projeto Existente

7.1 Modificações na Classe Jogador

Vamos estender a classe `Jogador` para suportar o novo sistema de movimentação:

Python

```python
# Adições à classe Jogador existente
class Jogador:
    def __init__(self, nome, idade, posicao):
        # ... atributos existentes ...

        # Novos atributos para movimentação
        self.posicao_atual = (0.0, 0.0)  # Posição atual no campo (x, y)
        self.posicao_objetivo = (0.0, 0.0)  # Onde o jogador quer estar
        self.posicao_tatica_base = (0.0, 0.0)  # Posição base na formação
        self.velocidade_atual = (0.0, 0.0)  # Vetor velocidade atual
        self.modo_movimento = "normal"  # "normal", "pressing", "corrida"

        # Histórico de posições para análise
        self.historico_posicoes = []
        self.distancia_percorrida = 0.0

    def definir_posicao_tatica_base(self, formacao: FormacaoTatica, indice:
int = 0):
        """Define a posição base do jogador na formação tática."""
        self.posicao_tatica_base = formacao.obter_posicao_base(self.posicao,
indice)
        self.posicao_atual = self.posicao_tatica_base
        self.posicao_objetivo = self.posicao_tatica_base

    def atualizar_historico_posicao(self):
        """Mantém histórico das últimas 100 posições para análise."""
        self.historico_posicoes.append(self.posicao_atual)
        if len(self.historico_posicoes) > 100:
            self.historico_posicoes.pop(0)

    def calcular_distancia_percorrida_tick(self, posicao_anterior:
Tuple[float, float]) -> float:
        """Calcula a distância percorrida no último tick."""
        x_ant, y_ant = posicao_anterior
        x_atual, y_atual = self.posicao_atual
        distancia = math.sqrt((x_atual - x_ant)**2 + (y_atual - y_ant)**2)
        self.distancia_percorrida += distancia
        return distancia
```

7.2 Modificações na Classe Time

A classe `Time` também precisa de atualizações para suportar formações mais dinâmicas:

Python

```python
class Time:
    def __init__(self, nome: str):
```

```python
        # ... atributos existentes ...

        # Novos atributos para movimentação
        self.formacao_atual = None
        self.formacoes_disponiveis = {
            "4-4-2": FORMACAO_442,
            "4-3-3": FORMACAO_433,
            "3-5-2": FORMACAO_352
        }
        self.linha_defensiva_atual = 20.0
        self.linha_ofensiva_atual = 60.0
        self.largura_time = 40.0  # Quão "aberto" o time joga

    def definir_formacao(self, nome_formacao: str):
        """Define a formação tática e posiciona os jogadores."""
        if nome_formacao in self.formacoes_disponiveis:
            self.formacao_atual = self.formacoes_disponiveis[nome_formacao]

            # Posicionar jogadores nas posições base
            contadores_posicao = {}
            for jogador in self.jogadores:
                posicao = jogador.posicao
                if posicao not in contadores_posicao:
                    contadores_posicao[posicao] = 0

                jogador.definir_posicao_tatica_base(self.formacao_atual,
contadores_posicao[posicao])
                contadores_posicao[posicao] += 1

    def ajustar_compactacao(self, nivel_compactacao: float):
        """
        Ajusta o quão compacto o time joga (0.0 = muito aberto, 1.0 = muito
fechado).
        """
        for jogador in self.jogadores:
            x_base, y_base = jogador.posicao_tatica_base

            # Ajustar Y (largura) baseado na compactação
            centro_campo_y = 30.0
            distancia_centro = y_base - centro_campo_y
            nova_distancia = distancia_centro * (1.0 - nivel_compactacao *
0.5)
            novo_y = centro_campo_y + nova_distancia

            jogador.posicao_tatica_base = (x_base, novo_y)
```

8. Conceitos-Chave e Aplicações Práticas

8.1 Algoritmos de Pathfinding Aplicados

O pathfinding no futebol é único porque:

1. **Obstáculos Dinâmicos**: Outros jogadores se movem constantemente
2. **Múltiplos Objetivos**: Um jogador pode ter vários destinos possíveis
3. **Restrições Táticas**: Não pode quebrar a formação drasticamente
4. **Tempo Real**: Decisões devem ser tomadas rapidamente

8.2 Coordenação Multi-Agente

Nosso sistema implementa coordenação multi-agente onde:

- Cada jogador é um agente autônomo
- Agentes se comunicam através do estado compartilhado do jogo
- Decisões individuais são influenciadas por objetivos coletivos
- Emergem comportamentos complexos da interação simples entre agentes

8.3 Otimização de Performance

Para manter o simulador rodando suavemente com 22 jogadores se movendo simultaneamente:

Python

```
class OtimizadorMovimentacao:
    def __init__(self):
        self.cache_calculos = {}
        self.ultimo_update_cache = 0

    def calcular_movimento_otimizado(self, jogador: Jogador, estado:
EstadoPartida):
        """
        Versão otimizada que usa cache para cálculos pesados.
        """
        # Usar cache para cálculos que não mudam a cada tick
        chave_cache = f"{jogador.id}_{int(estado.tempo_atual // 1.0)}"

        if chave_cache in self.cache_calculos:
            return self.cache_calculos[chave_cache]

        # Calcular movimento (lógica anterior)
```

```
        resultado = self._calcular_movimento_completo(jogador, estado)

        # Armazenar no cache
        self.cache_calculos[chave_cache] = resultado

        # Limpar cache antigo a cada 10 segundos
        if estado.tempo_atual - self.ultimo_update_cache > 10.0:
            self._limpar_cache_antigo(estado.tempo_atual)
            self.ultimo_update_cache = estado.tempo_atual

        return resultado
```

Exercícios Práticos

Exercício 1: Implementar Movimentação de Meio-Campo

Modifique a função `calcular_movimento_meia` para incluir:

- Busca por espaços livres entre as linhas adversárias
- Movimento de apoio quando companheiros estão pressionados
- Cobertura defensiva quando necessário

Exercício 2: Sistema de Impedimento

Implemente um sistema que:

- Calcule a linha de impedimento em tempo real
- Ajuste a movimentação dos atacantes para evitar impedimento
- Permita movimentos de quebra de impedimento no momento certo

Exercício 3: Movimentação em Bolas Paradas

Crie comportamentos específicos para:

- Escanteios (posicionamento na área)
- Faltas (barreira e marcação)
- Lateral (opções de passe)

Conclusão

A movimentação dinâmica e o posicionamento tático são fundamentais para criar um simulador de futebol realista. Nesta aula, implementamos:

- Algoritmos de pathfinding adaptados para futebol
- Sistemas de coordenação entre jogadores
- Comportamentos específicos por posição
- Integração com o motor de jogo existente

Na próxima aula, vamos abordar a física da bola e como ela interage com os jogadores em movimento, adicionando ainda mais realismo ao nosso simulador.

Professor: Manus AI

Curso: Simulador de Futebol em Python - Parte 2

Próxima Aula: Física da Bola e Interações Avançadas