Futebol de Robôs
Trabalho 2 – Programação de Baixo Nível

Fernanda Franceschini[1], Luana Sostisso[2]

Escola Politécnica— PUCRS

29 de maio de 2024

Resumo

Este trabalho visa apresentar uma solução para o problema proposto na disciplina de Programação de Baixo Nível no terceiro semestre. O problema consiste em desenvolver um programa que simula um jogo de futebol de robôs em C, aplicando conceitos avançados de programação. A simulação inclui três tipos de robôs: goleiro, defensor e atacante, cada um com comportamentos específicos. A simulação pode terminar ao atingir um número específico de gols ou após um tempo determinado. O programa deve gerenciar as ações dos robôs, a interação com a bola e a pontuação do jogo.

1. Introdução

A simulação de futebol de robôs é um desafio intrigante que combina aspectos de controle autônomo e programação de baixo nível. Este trabalho, desenvolvido no âmbito da disciplina de Programação de Baixo Nível, tem como objetivo praticar e aplicar conceitos avançados da linguagem C através da criação de um programa que simula uma partida de futebol entre robôs. Nesta simulação, três tipos de robôs são utilizados: goleiro, defensor e atacante, cada um com comportamentos definidos para garantir um jogo dinâmico e estratégico. O goleiro é responsável por proteger a área do gol, enquanto os defensores têm a tarefa de marcar os atacantes adversários e proteger a grande área. Os atacantes, por sua vez, são encarregados de perseguir a bola e tentar marcar gols, respeitando certas restrições para evitar jogadas desastrosas perto de seu próprio gol.

Cada time é composto por um goleiro, três defensores e dois atacantes, e a partida é vencida pelo time que marcar mais gols. O jogo acaba quando um time fizer no mínimo 3 gols e tiver uma diferença de 2 gols do time adversário. Este trabalho visa não apenas criar uma simulação funcional, mas também demonstrar habilidades avançadas em programação em C, como controle de fluxo e interação entre múltiplos componentes de um sistema complexo.

2. Processo de Solução

2.1 Função Bola

No desenvolvimento inicial do programa em C, decidiu-se criar funções genéricas para controlar a ação de cada agente (uma para atacantes, outra para zagueiros, uma para os goleiros e uma para a bola). De partida, foi implementado um algoritmo para a movimentação da bola. Este algoritmo é crucial, pois a posição e o movimento da bola influenciam diretamente as ações dos robôs jogadores em campo. A função *moveBola* precisa apresentar três funcionalidades: setar a velocidade da bola e a posição da bola, além de verificar quando a bola entrar no gol para marcar no placar.

```
void moveBola(cpBody* body, void* data){
    cpVect velBola = cpvclamp(cpBodyGetVelocity(body), 55);
    cpBodySetVelocity(body, velBola);
```

```
    cpVect ballPos  = cpBodyGetPosition(body);
    if(ballPos.x < 45 && ballPos.y > 326 && ballPos.y < 386){
        score2++;
        resetPositionPlayers();
    }
    if(ballPos.x > 978 && ballPos.y > 326 && ballPos.y < 386){
        score1++;
        resetPositionPlayers();
    }
    isTheGameOver();
}
```

A função *moveBola* inicia obtendo a velocidade atual da bola utilizando *cpBodyGetVelocity*(body), que retorna um vetor de velocidade. A seguir, a velocidade da bola é ajustada com a função cpvclamp()), que limita a magnitude do vetor de velocidade ao valor máximo especificado, garantindo que a bola não se mova mais rápido do que o permitido, contribuindo para uma simulação realista. Finalmente, a velocidade ajustada é aplicada de volta à bola com *cpBodySetVelocity(body, velBola),* garantindo que qualquer ajuste necessário na velocidade é realizado.

Em seguida, a função obtém a posição atual da bola com *cpBodyGetPosition(body),* que retorna um vetor representando a posição (x, y) da bola no campo. Esta posição é crucial para determinar se a bola entrou na área do gol. O código então verifica se a posição da bola atende aos critérios de um gol, começando pela goleira esquerda e depois pela direita. Se a coordenada x da bola e a coordenada y estiverem dentro dos parâmetros que é considerada a goleira, neste caso, o score do time que fez gol é incrementado e a bola é reposicionada no centro do campo pela função *resetPositionPlayers()*.

```
void resetPositionPlayers () {

    cpBodySetPosition(goleiroEsquerda, golEsq);

    cpBodySetPosition(zagueiroEsquerda1, defeEsq1);

    cpBodySetPosition(zagueiroEsquerda2, defeEsq2);

    cpBodySetPosition(zagueiroEsquerda3, defeEsq3);

    cpBodySetPosition(atacanteEsquerda1, ataEsq1);

    cpBodySetPosition(atacanteEsquerda2, ataEsq2);

    cpBodySetPosition(goleiroDireita, golDir);

    cpBodySetPosition(zagueiroDireita1, defeDir1);

    cpBodySetPosition(zagueiroDireita2, defeDir2);

    cpBodySetPosition(zagueiroDireita3, defeDir3);
```

```
        cpBodySetPosition(atacanteDireita1, ataDir1);

        cpBodySetPosition(atacanteDireita2, ataDir2);

        cpBodySetVelocity(ballBody, cpv(0,0));

        cpBodySetPosition(ballBody, centroDoCampo);

    }
```

Esta função é responsável por retornar os jogadores as suas posições inicias e a bola ao meio de campo com a velocidade zerada *(cpv(0,0))*, preparando a bola para o próximo lance. No final do código anterior, a função *isTheGameOver()* é chamada para verificar se o jogo terminou de acordo com a regra determinada.

```
    void isTheGameOver () {

          if((score1 >= 3 && (score1-score2)>2) || (score2 >= 3 &&(score2-
    score1)>2)){

                    gameOver = 1;

            }

    }
```

Se a quantidade de gols for maior ou igual que 3 e houver uma diferença de dois gols entre cada time, o jogo termina.

2.2 Função Goleiro

Após incrementar o código da bola, o próximo passo é a implementação dos goleiros. Esse método é responsável por controlar a movimentação dos goleiros, garantindo que eles protejam suas respectivas goleiras e reajam adequadamente à posição da bola. A seguir, apresentamos a implementação detalhada

```
void moveGoleiro(cpBody* body, void* data){
    UserData* ud = (UserData*)data;
    jogador_data j = ud->jogadorData;


    checkAndApplyBoundaryImpulse(body, data);


    cpVect vel = cpBodyGetVelocity(body);
    vel = cpvclamp(vel, 10);
    cpBodySetVelocity(body, vel);


    cpVect robotPos = cpBodyGetPosition(body);
    cpVect ballPos  = cpBodyGetPosition(ballBody);
```

```
    cpVect pos = robotPos;
    pos.x = -robotPos.x;
    pos.y = -robotPos.y;
    cpVect delta = cpvmult(cpvnormalize(cpvadd(ballPos,robotPos)),20);

    cpVect posicao_goleira;
    int valor_grande_area;
    if (j.lado == LEFT) {
        posicao_goleira = golEsq;
        valor_grande_area = 189;
    } else {
        posicao_goleira = golDir;
        valor_grande_area = 835;
    }

    bool in_grande_area = (ballPos.x < valor_grande_area && j.lado == LEFT) ||
(ballPos.x > valor_grande_area && j.lado == RIGHT);
    bool in_valid_y_range = (ballPos.y < 533 && ballPos.y > 180);

    if (in_grande_area && in_valid_y_range) {
        cpBodyApplyImpulseAtWorldPoint(body, delta, robotPos);
    } else {
           cpBodyApplyImpulseAtWorldPoint(body, cpvadd(posicao_goleira, pos),
posicao_goleira);
         if (fabs(robotPos.x - posicao_goleira.x) < 0.1 && fabs(robotPos.y -
posicao_goleira.y) < 0.1) {
            cpBodySetVelocity(body, cpv(0,0));
        }
    }
}
```

Inicialmente, o método começa verificando e aplicando limites de impulso usando *checkAndApplyBoundaryImpulse().* Seguindo a lógica da função da bola, obtemos a velocidade atual do goleiro com *cpBodyGetVelocity(body)* e ajustando-a para que não exceda um limite máximo com *cpvclamp(vel,10))*. A velocidade ajustada é então aplicada ao goleiro com o método *cpBodySetVelocity(body, vel)*. Em seguida, as posições atuais do goleiro *(robotPos)* e da bola *(ballPos)* são obtidas. Um vetor *pos* é criado com as coordenadas de *robotPos* e somado à posição da bola para calcular o *delta*, que representa a direção do impulso, ajudando a garantir que o jogador se mova na direção da bola sem se preocupar com a magnitude inicial do vetor. Isso é feito usando *cpvnormalize(delta),* garantindo que o impulso tenha sempre a mesma direção.

Como é uma função genérica, é preciso verificar qual o goleiro estamos modificando para setar as suas posições corretamente. Dependendo do lado do campo em que o goleiro está (esquerdo ou direito), a

posição da goleira e o valor da grande área são definidos. Para aplicar essa lógica, foi criado um *if* que verificar se o jogador *"j"* é do lado direito ou esquerdo, e então determinar qual o lado da sua goleira e qual o valor da grande área(área que o goleiro se movimenta). A função então analisa se a bola está dentro da grande área e em uma faixa válida de y; se estiver, um impulso é aplicado ao goleiro para movê-lo em direção à bola. Caso contrário, um impulso é aplicado para retornar o goleiro à sua posição inicial perto da goleira. Se o goleiro já estiver próximo da posição inicial, sua velocidade é zerada *cpBodySetVelocity(body, cpv(0,0)),* para mantê-lo estático. Este algoritmo assegura que o goleiro atua defensivamente dentro dos limites de sua área designada, respondendo eficazmente às posições dinâmicas da bola.

2.3 Função Defensores

Para completar a implementação do jogo de futebol de robôs, além dos goleiros, é necessário desenvolver funções para controlar os atacantes e os zagueiros. A seguir é a implementação dos zagueiros.

```
void moveDefensor (cpBody* body, void* data){
    UserData* ud = (UserData*)data;
    jogador_data j = ud->jogadorData;

    cpVect jogador_posicao = j.pos_inicial;

    checkAndApplyBoundaryImpulse(body, data);

    cpVect vel = cpBodyGetVelocity(body);
    vel = cpvclamp(vel, 28);
    cpBodySetVelocity(body, vel);

    // Obtém a posição do robô e da bola...
    cpVect robotPos = cpBodyGetPosition(body);
    cpVect ballPos  = cpBodyGetPosition(ballBody);

    cpVect pos = robotPos;
    pos.x = -robotPos.x;
    pos.y = -robotPos.y;
    cpVect delta = cpvadd(ballPos,pos);

    if (cpvdist(robotPos, ballPos) < 25) { // os defensores estavam muito
bons, reduzimos o raio de ação
        srand(time(NULL));
        int sorteio = rand() % 2;
```

```
                aplicarImpulsoNaBola(obterPosicaoAtacante(j.lado,  sorteio),
ballPos);
    }

    //chega perto da bola?
    int valores[2];
    if (j.pos_inicial.y == 180) {
        valores[0] = 0;
        valores[1] = 237;
    } else if (j.pos_inicial.y == 356) {
        valores[0] = 237;
        valores[1] = 474;
    } else {
        valores[0] = 474;
        valores[1] = 712;
    }

    bool isLeft = (j.lado == LEFT);
    bool isBallInZone = (ballPos.y > valores[0] && ballPos.y < valores[1]);
    bool isLeftZone = (ballPos.x <= 512);
    bool isRightZone = (ballPos.x > 512);

    if ((isLeft && isLeftZone && isBallInZone) || (!isLeft && isRightZone
&& isBallInZone)) {
        cpBodyApplyImpulseAtWorldPoint(body, delta, robotPos);
    } else {
        cpVect deltaIni = cpvadd(jogador_posicao, pos);
        cpBodyApplyImpulseAtWorldPoint(body, deltaIni, jogador_posicao);
        if (fabs(robotPos.x - jogador_posicao.x) < 0.1 && fabs(robotPos.y -
jogador_posicao.y) < 0.1) {
            cpBodySetVelocity(body, cpv(0, 0));
        }
    }
}
```

Para não tornar este artigo muito repetitivo, percebe-se que o início de todas as funções são praticamente idênticas, sem a necessidade de explica-las novamente. O diferencial que vemos nessa função é a logica utilizada para verificar a distância entre o defensor e a bola *(cpvdist(robotPos, ballPos))*. Se esta for menor que 25 unidades, um atacante é escolhido aleatoriamente *(rand() % 2)*

para receber um impulso na direção da bola. O método *aplicarImpulsoNaBola* é responsável por realizar essa ação.

```
void aplicarImpulsoNaBola(cpVect mira, cpVect ballPos) {
    srand(time(NULL));
    int sorteio = rand() % 51;
    mira.x -= sorteio;
    cpVect chute = cpvmult(cpvnormalize(cpvsub(mira, ballPos)), 7);
    cpBodyApplyImpulseAtWorldPoint(ballBody, chute, ballPos);
}
```

Além disso, em seguida determina-se dinamicamente os limites verticais da zona de ação de um defensor com base em sua posição inicial no campo de jogo. A variável *j.pos_inicial.y* indica a posição vertical inicial do defensor, com base nesse valor o código atribui valores específicos ao array valores, que define os limites verticais da zona onde o defensor considera estar próximo o suficiente da bola para agir diretamente sobre ela. Isso é fundamental para decidir se o defensor deve tentar interceptar a bola ou se deve se posicionar estrategicamente, dependendo da posição da bola no campo. Essa abordagem ajuda a otimizar o comportamento dos defensores, ajustando dinamicamente sua resposta com base em sua posição inicial no campo de jogo.

No final do código, é verificado se a bola está na zona de ação do defensor e no lado correto do campo. Se sim, um impulso é aplicado diretamente em direção à *bola cpBodyApplyImpulseAtWorldPoint.* Caso contrário, o defensor retorna à sua posição inicial *(jogador_posicao*) usando *cpBodyApplyImpulseAtWorldPoint* e sua velocidade é zerada se estiver muito próxima da posição inicial.

2.4 Função Atacante

Por fim, temos a função que movimenta os atacantes.

```
void moveAtacante(cpBody* body, void* data) {
    UserData* ud = (UserData*)data;
    jogador_data j = ud->jogadorData;
    cpVect jogador_posicao = j.pos_inicial;

    checkAndApplyBoundaryImpulse(body, data);

    cpVect vel = cpvclamp(cpBodyGetVelocity(body), 30);
    cpBodySetVelocity(body, vel);

    cpVect ballPos = cpBodyGetPosition(ballBody);
    cpVect robotPos = cpBodyGetPosition(body);

    cpVect pos = robotPos;
    pos.x = -robotPos.x;
```

```
 pos.y = -robotPos.y;

 cpVect delta = cpvmult(cpvnormalize(cpvadd(ballPos, cpvneg(robotPos))),
20);

 cpVect posicao_goleira;
 cpVect posicaoOutroAtacante;

 if (j.lado == LEFT) {
     posicao_goleira = golDir;
            posicaoOutroAtacante   =   (body   ==   atacanteEsquerda1)   ?
obterPosicaoAtacante(j.lado, 1) : obterPosicaoAtacante(j.lado, 0);
 } else {
     posicao_goleira = golEsq;
             posicaoOutroAtacante   =   (body   ==   atacanteDireita1)   ?
obterPosicaoAtacante(j.lado, 1) : obterPosicaoAtacante(j.lado, 0);
 }

 cpFloat distanciaParaBola = cpvdist(robotPos, ballPos);
 cpFloat distanciaParaGol = cpvdist(robotPos, posicao_goleira);
  cpFloat  distanciaOutroAtacanteParaGol  =  cpvdist(posicaoOutroAtacante,
posicao_goleira);

 if (distanciaParaGol > distanciaOutroAtacanteParaGol && distanciaParaBola
<28) {
     aplicarImpulsoNaBola(posicaoOutroAtacante, ballPos);
 } else if (distanciaParaBola < 32) {
     aplicarImpulsoNaBola(posicao_goleira, ballPos);
 }

 bool isLeft = (j.lado == LEFT);
     cpBool   isCloser   =   cpvdist(posicaoOutroAtacante,   ballPos)   >
distanciaParaBola;

 if ((isLeft && ballPos.x > 430) || (!isLeft && ballPos.x < 596)) {
     if (isCloser) {
         cpBodyApplyImpulseAtWorldPoint(body, delta, robotPos);
     }
 } else {
     cpVect deltaIni = cpvadd(jogador_posicao, pos);
     cpBodyApplyImpulseAtWorldPoint(body, deltaIni, jogador_posicao);
```

```
            if (fabs(robotPos.x - jogador_posicao.x) < 0.1 && fabs(robotPos.y -
        jogador_posicao.y) < 0.1) {
                cpBodySetVelocity(body, cpv(0, 0));
            }
         }
      }
```

O diferencial dessa função para as outras é que ela contém a definição das variáveis *posicao_goleira* e *posicaoOutroAtacante* com base no lado de jogo (*j.lado*).

```
      cpVect obterPosicaoAtacante(lado lado, int atacante) {

          switch (lado) {

              case RIGHT: return (atacante == 0) ? cpBodyGetPosition(atacanteDi-
reita1) : cpBodyGetPosition(atacanteDireita2);

              case LEFT: return (atacante == 0) ? cpBodyGetPosition(atacanteEs-
querda1) : cpBodyGetPosition(atacanteEsquerda2);

              default: return cpvzero;

          }

      }
```

Se o lado for `LEFT`, o *`posicao_goleira`* é definido como *`golDir`*, representando a posição da goleira adversária, e *`posicaoOutroAtacante`* é escolhido entre `atacanteEsquerda1` e `atacanteEsquerda2`, dependendo do corpo (`body`) que está sendo movido. Se o lado for RIGHT, as posições são atribuídas de forma similar, com *posicao_goleira* sendo *golEsq* e *posicaoOutroAtacante* sendo um dos atacantes direitos.

Em seguida, o código calcula a distância do robô *robotPos* para a bola *ballPos*, para a posição da goleira *posicao_goleira* e para o outro atacante *posicaoOutroAtacante*. Com base nessas distâncias, ele decide se o atacante deve passar a bola para o outro atacante mais próximo do gol ou se deve chutar diretamente para o gol. Se a distância até o gol *distanciaParaGol* for maior do que a distância do outro atacante até o gol *distanciaOutroAtacanteParaGol* e a distância para a bola *distanciaParaBola* for menor que 28 unidades, o atacante aplica um impulso na bola na direção do outro atacante. Caso contrário, se a distância para a bola for menor que 32 unidades, o atacante chuta em direção à posição da goleira *posicao_goleira*.

Além disso, o código verifica se o atacante está na posição correta em relação à bola para decidir se deve se mover em direção a ela (delta) ou se deve retornar à sua posição inicial *jogador_posicao*. Isso é determinado pela posição da bola *ballPos.x* em relação ao campo e se o outro atacante está mais próximo dela pela variável *isCloser*. Se as condições forem atendidas, um impulso é aplicado no robô na direção da bola para avançar na jogada. Caso contrário, o robô se move de volta à sua posição inicial, com sua velocidade sendo ajustada para zero quando ele está próximo o suficiente dessa posição. Essa lógica ajuda a otimizar o movimento dos atacantes, priorizando ações estratégicas com base na posição da bola e dos jogadores em campo.

2.5 Limpar memória alocada dinamicamente

A função freeCM() tem a responsabilidade de liberar a memória ocupada por todos os corpos físicos (jogadores, bola e paredes), formas associadas a esses corpos e o ambiente de física (espaço). É essencial para garantir que todos os recursos alocados durante a execução do programa sejam devidamente liberados ao final, evitando vazamento de memória e mantendo a integridade do sistema.

```
void freeCM(){

    printf("Cleaning up!\n");

    UserData* ud = cpBodyGetUserData(ballBody);

    cpShapeFree(ud->shape);

    cpBodyFree(ballBody);

    ud = cpBodyGetUserData(goleiroEsquerda);

    cpShapeFree(ud->shape);

    cpBodyFree(goleiroEsquerda);

    ud = cpBodyGetUserData(zagueiroEsquerda1);

    cpShapeFree(ud->shape);

    cpBodyFree(zagueiroEsquerda1);

    ud = cpBodyGetUserData(zagueiroEsquerda2);

    cpShapeFree(ud->shape);

    cpBodyFree(zagueiroEsquerda2);

    ud = cpBodyGetUserData(zagueiroEsquerda3);

    cpShapeFree(ud->shape);

    cpBodyFree(zagueiroEsquerda3);

    ud = cpBodyGetUserData(atacanteEsquerda1);

    cpShapeFree(ud->shape);

    cpBodyFree(atacanteEsquerda1);

    ud = cpBodyGetUserData(atacanteEsquerda2);

    cpShapeFree(ud->shape);

    cpBodyFree(atacanteEsquerda2);

    ud = cpBodyGetUserData(goleiroDireita);

    cpShapeFree(ud->shape);
```

```
                cpBodyFree(goleiroDireita);
                ud = cpBodyGetUserData(zagueiroDireita1);
                cpShapeFree(ud->shape);
                cpBodyFree(zagueiroDireita2);
                ud = cpBodyGetUserData(zagueiroDireita2);
                cpShapeFree(ud->shape);
                cpBodyFree(zagueiroDireita2);
                ud = cpBodyGetUserData(zagueiroDireita3);
                cpShapeFree(ud->shape);
                cpBodyFree(zagueiroDireita3);
                ud = cpBodyGetUserData(atacanteDireita1);
                cpShapeFree(ud->shape);
                cpBodyFree(atacanteDireita1);
                ud = cpBodyGetUserData(atacanteDireita2);
                cpShapeFree(ud->shape);
                cpBodyFree(atacanteDireita2);
                cpShapeFree(travessaoD1);
                cpShapeFree(travessaoD2);
                cpShapeFree(travessaoE1);
                cpShapeFree(travessaoD1);
                cpShapeFree(leftWall);
                cpShapeFree(rightWall);
                cpShapeFree(topWall);
                cpShapeFree(bottomWall);
                cpSpaceFree(space);
}
```

3. Conclusão

Concluir a implementação de uma simulação de futebol de robôs em C envolve enfrentar algumas dificuldades técnicas e conceituais significativas. Primeiramente, é crucial lidar com a complexidade de controlar múltiplos agentes (goleiros, atacantes e zagueiros) em um ambiente físico simulado. Cada tipo de jogador requer algoritmos específicos para movimentação, decisões táticas e interação com a bola e outros jogadores, oque gerou uma grande dificuldade para testarmos os funcionamentos de cada objeto separadamente visto que o movimento de um agente depende dos outros e da bola.

Além disso, outras dificuldades incluíram compreender as táticas adequadas a serem usadas, entender o funcionamento de um jogo de futebol e determinar quais regras deveriam ser aplicadas, o que demandou tempo considerável. Uma ideia era fazer com que os dois jogadores mais próximos da bola se movimentassem até ela, inicialmente parecia uma estratégia promissora para maximizar a eficiência e o controle sobre a bola no jogo. No entanto, essa abordagem revelou desafios práticos que comprometeram sua viabilidade e eficácia durante a implementação. O futebol de robôs valoriza a autonomia individual de cada jogador para tomar decisões rápidas e adaptar-se às condições em constante mudança do jogo, tentar fazer com que um jogador controle diretamente o movimento de outro limitou essa flexibilidade, tornando mais difícil para a equipe responder de maneira eficaz. Assim, a estratégia de movimentar os jogadores quando a bola está na região deles (área do campo onde cada jogador é designado para atuar com base na posição inicial definida estrategicamente) foi mantida como a abordagem principal para coordenar as ações da equipe de futebol de robôs.

Outro ponto de dificuldade enfrentado foi a incorporação das figuras PNG no program, exigindo a manipulação precisa dos arquivos de imagem para garantir que fossem exibidos corretamente e de maneira eficiente durante a simulação. Junto dessas complexidades, determinar a posição precisa dos jogadores e calcular o impulso necessário para mover ou chutar a bola de maneira eficaz. Esse desafio envolveu cálculos vetoriais e transformações de coordenadas para posicionar os jogadores corretamente no campo, e calcular os impulsos considerando a física do movimento para simular de forma realista o comportamento dos robôs. Ajustar os parâmetros de força e direção dos impulsos exigiu um processo de tentativa e erro.

Por fim, o desenvolvimento de um projeto como este geralmente envolve a coordenação de várias partes do sistema, a lógica de simulação, a física do jogo e a representação visual dos movimentos dos jogadores e da bola. Integrar essas partes de forma coesa e garantir que todas operem em sincronia pode ser um desafio técnico significativo. Assim, enquanto implementar uma simulação de futebol de robôs em C oferece oportunidades para explorar conceitos avançados de programação, física computacional e estratégia de jogo, também o desenvolvimento do programa demonstrou a importância da persistência e do método iterativo para alcançar resultados satisfatórios. A experiência adquirida não apenas ampliou o conhecimento prático, mas também reforçou a capacidade de enfrentar e resolver desafios complexos em projetos de desenvolvimento de software.