Trabalho de Conclusão de Curso

Visual servoing em ambientes virtuais com Unity 3D

de **Bruno Lemos de Lima**

orientado por
Prof. Dr. Glauber Rodrigues Leite

Universidade Federal de Alagoas
Instituto de Computação
Maceió, Alagoas
6 de janeiro de 2025

UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação

VISUAL SERVOING EM AMBIENTES VIRTUAIS COM UNITY 3D

Trabalho de Conclusão de Curso submetido ao Instituto de Computação da Universidade Federal de Alagoas como requisito parcial para a obtenção do grau de Engenheiro de Computação.

Bruno Lemos de Lima

Orientador: Prof. Dr. Glauber Rodrigues Leite

Banca Avaliadora:

Allan de Medeiros Martins	Prof. Dr., UFRN
Andressa Martins Oliveira	Prof. Me., UFAL
Ícaro Bezerra Queiroz de Araújo	Prof. Dr., UFAL

Maceió, Alagoas
6 de janeiro de 2025

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

L732v Lima, Bruno Lemos de.
Visual servoing em ambientes virtuais com Unity 3D / Bruno Lemos de Lima. – 2024.
50 f. : il.

Orientador: Glauber Rodrigues Leite.
Monografia (Trabalho de conclusão de curso em Engenharia de Computação) - Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2024.

Bibliografia: f. 49-50.

1. *Visual servoing*. 2. Robótica. 3. Unity 3D (Recurso eletrônico). 4. ROS 2 - Comutação por pacotes (Transmissão de dados). 5. ROS-TCP-Connector - Comutação por pacotes (Transmissão de dados). I. Título.

CDU: 004.896

Dedicatória

Dedico este trabalho, em primeiro lugar, à minha família, cujo apoio incondicional e incentivo constante tornaram este momento possível. A vocês, minha eterna gratidão.
Em segundo lugar, dedico à Universidade Federal de Alagoas (UFAL), cuja comunidade acadêmica me proporcionou um ambiente de aprendizado e crescimento, sendo fundamental para a minha formação e para a concretização deste trabalho.
Por fim, dedico a você, leitor, que compartilha o interesse pela robótica e pela inovação.

Agradecimentos

Em primeiro lugar, expresso minha gratidão à minha família. Ao meu pai, Benildo, e à minha mãe, Gilvânia, pelo amor, apoio e incentivo incondicionais ao longo de toda a minha formação pessoal e acadêmica. À minha irmã, Bruna, pelo carinho e companhia durante essa jornada.

Agradeço especialmente à minha namorada, Vívian, cujo apoio, paciência e carinho foram fundamentais ao longo de todo o percurso. Seu companheirismo me fortaleceu nos momentos mais desafiadores, e sua presença tornou cada etapa dessa caminhada mais leve e gratificante.

Aos professores do curso de Engenharia da Computação da Universidade Federal de Alagoas, manifesto meu profundo reconhecimento pelo conhecimento transmitido e pela orientação ao longo da minha formação. Em particular, agradeço aos professores Thiago Cordeiro, Márcio Ribeiro, Ícaro Bezerra, Erick Barboza, Glauber Leite e Andressa Martins, cujas valiosas contribuições foram essenciais para o desenvolvimento deste trabalho e para minha trajetória acadêmica. A todos, meus mais sinceros agradecimentos.

Aos meus amigos Karla, Pedro, Neto, Fernanda, Lilian, Lucas, Eduarda, Rodrigo, Arthur, Hugo, Kevin, Álvaro, Júlia e Laryssa, que tornaram essa jornada mais leve e agradável. A presença de vocês, seja nos momentos de estudo, trabalho ou descontração, foi essencial para superar os desafios e chegar até aqui.

Por fim, expresso minha sincera gratidão a todos que, de alguma forma, contribuíram para a realização deste trabalho, seja direta ou indiretamente. O apoio de cada um foi imprescindível para que eu alcançasse esta conquista.

Science isn't about why! It's about why not! – Cave Johnson, Portal 2

Resumo

A implementação de algoritmos de *visual servoing* enfrenta desafios como a escassez de robôs físicos para experimentação e a complexidade na integração de sistemas. Este trabalho apresenta o desenvolvimento de um simulador em ambiente virtual para visualização e controle de robôs, focando na validação de algoritmos de controle visual. A simulação é implementada na plataforma Unity 3D, integrada ao ROS 2 por meio do pacote ROS-TCP-Connector, permitindo a troca de dados via protocolo TCP/IP. O simulador coleta dados da câmera acoplada ao efetuador do robô, como imagens RGB e mapas de profundidade, além de informações sobre o estado das juntas, incluindo posições, velocidades e esforços. Ele também possibilita o controle individual das juntas, promovendo flexibilidade no desenvolvimento de algoritmos. Embora o foco seja o *visual servoing*, o simulador é versátil e pode ser aplicado em outras abordagens de controle. Adicionalmente, foi desenvolvido um pacote ROS 2 contendo um algoritmo de *visual servoing* para controle automatizado do robô na simulação. A proposta oferece um ambiente para validar algoritmos de *visual servoing* em cenários simulados, contribuindo para o desenvolvimento de soluções na área de controle visual.

***Palavras-chave*: *Visual Servoing*; *Simulação Robótica*; *Unity 3D*; *ROS 2*; *ROS-TCP-Connector*.**

Abstract

The implementation of visual servoing algorithms faces challenges such as the scarcity of physical robots for experimentation and the complexity of system integration. This work presents the development of a simulator in a virtual environment for robot visualization and control, focusing on validating visual control algorithms. The simulation is implemented on the Unity 3D platform, integrated with ROS 2 through the ROS-TCP-Connector package, enabling data exchange via the TCP/IP protocol. The simulator collects data from the camera attached to the robot's end-effector, such as RGB images and depth maps, as well as joint state information, including positions, velocities, and efforts. It also allows individual joint control, promoting flexibility in algorithm development. Although the focus is on visual servoing, the simulator is versatile and can be applied to other control approaches. Additionally, a ROS 2 package containing a visual servoing algorithm was developed for automated robot control within the simulation. The proposal provides an environment to validate visual servoing algorithms in simulated scenarios, contributing to the development of solutions in the field of visual control.

***Keywords*: *Visual Servoing*; *Robotic Simulation*; *Unity 3D*; *ROS 2*; *ROS-TCP-Connector*.**

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

T Matriz de transformação homogênea.

q Vetor de coordenadas articulares do robô.

J Matriz Jacobiana.

p Coordenadas de um ponto no plano da imagem.

f Distância focal da câmera.

λ Ganho de controle proporcional.

Z Profundidade do ponto no espaço da câmera.

$\dot{q}$ Velocidades articulares do robô.

$\dot{x}$ Velocidades lineares e angulares no espaço operacional.

ω Velocidade angular do efetuador final.

v Velocidade linear do efetuador final.

A_i Matriz de transformação homogênea entre as juntas i e $i + 1$.

K Matriz intrínseca da câmera.

R Matriz de rotação da câmera no espaço tridimensional.

t Vetor de translação da câmera no espaço tridimensional.

Lista de Abreviaturas

ROS *Robot Operating System.*

IBVS *Image-Based Visual Servoing.*

PBVS *Position-Based Visual Servoing.*

DOF *Degrees of Freedom.*

FOV *Field of View.*

URDF *Unified Robot Description Format.*

TCP/IP *Transmission Control Protocol/Internet Protocol.*

RGB *Red, Green, Blue* (canais de cores).

DH Denavit-Hartenberg (parâmetros de cinemática).

HDRP *High Definition Render Pipeline.*

AO *Ambient Occlusion.*

SMC *Sliding Mode Control.*

SLAM *Simultaneous Localization and Mapping.*

Sumário

Capítulo 1

Introdução

A robótica moderna está presente em diversas áreas, desde a automação industrial até aplicações na saúde e serviços domésticos Craig (2016). Com o avanço tecnológico, cresce a demanda por sistemas robóticos mais inteligentes e adaptativos, capazes de interagir com ambientes complexos e dinâmicos Spong et al. (2020). Nesse contexto, o controle de robôs baseado em visão computacional, conhecido como *visual servoing*, é uma técnica que contribui para aprimorar a precisão e a autonomia desses sistemas Corke (2023).

O *visual servoing* utiliza informações visuais obtidas por sensores, como câmeras, para controlar os movimentos de um robô, permitindo que ele ajuste suas ações em resposta a mudanças no ambiente Corke (2023). Essa abordagem integra visão computacional, controle automático e cinemática robótica, resultando em sistemas capazes de realizar tarefas complexas com maior adaptabilidade, mesmo em cenários não estruturados.

O Filtro de Kalman é amplamente utilizado para estimar estados de sistemas dinâmicos em presença de ruídos e incertezas. No contexto de controle visual, Marshall and Lipkin (2014) introduziram uma lei de controle baseada no Filtro de Kalman que reduz a sensibilidade a ruídos no sistema de *visual servoing*. Essa abordagem permite aprimorar o desempenho do robô mesmo sem uma calibração precisa dos parâmetros do sistema, caracterizando um *uncalibrated visual servoing*. Além disso, técnicas como o Filtro de Kalman com critério de máxima correntropia regularizado (RMCKF) são eficazes para lidar com incertezas operacionais, incluindo ruído não gaussiano e mudanças ambientais significativas, permitindo que o robô adapte suas ações de maneira eficiente em condições desafiadoras Leite et al. (2023). No entanto, o desenvolvimento e teste de algoritmos de *visual servoing* enfrentam desafios, principalmente devido à disponibilidade limitada de plataformas físicas para experimentação e à complexidade da integração de múltiplos sistemas Siciliano et al. (2009).

A simulação em ambientes controlados é fundamental para a validação de algoritmos, especialmente quando a integração de múltiplos sistemas é necessária. Essa abordagem permite testar e refinar soluções antes da implementação em *hardware* real Lynch and Park (2017). Simuladores robóticos reproduzem cenários variados e permitem a valida-

ção de estratégias de controle em um ambiente seguro. No entanto, muitos simuladores enfrentam limitações em fidelidade visual, capacidade de lidar com problemas complexos e integração com sistemas de controle avançados, especialmente em aplicações que envolvem alto processamento de dados e algoritmos sofisticados Corke (2023).

Ferramentas como o Unity 3D têm ganhado destaque pela capacidade de criar ambientes virtuais altamente realistas e interativos, sendo eficazes em simulações robóticas que demandam alta fidelidade visual. Estudos recentes demonstram que o Unity 3D, integrado ao ROS por meio de pacotes como o ROS-TCP-Connector[1] e o ROS-TCP-Endpoint[2], oferece uma plataforma robusta para o desenvolvimento de sistemas robóticos avançados, especialmente em ambientes amplos ou com alta demanda de renderização gráfica. Comparativamente, o Unity 3D se destaca por sua capacidade de renderizar sombras e texturas detalhadas, melhorando a precisão de algoritmos baseados em visão, como SLAM, em relação a outras ferramentas de simulação robótica, como o Gazebo Platt and Ricks (2022).

Considerando o avanço na integração entre visão computacional e controle de robôs, este trabalho propõe desenvolver uma aplicação para simular o comportamento de um robô controlado por *visual servoing* em um ambiente virtual. Utilizando o Unity 3D para modelagem e renderização e o ROS 2 como *framework* para desenvolvimento de software de robôs, a aplicação integra captura de imagens e informações de sensores virtuais com algoritmos de *visual servoing*. Essa abordagem permite explorar a interação entre algoritmos de controle em tempo real e dados visuais, proporcionando uma plataforma para análise de desempenho, ajuste de parâmetros e experimentação. Serão explorados os aspectos de comunicação bidirecional entre os sistemas, a implementação dos algoritmos de controle e a validação da precisão e eficiência do sistema em diferentes cenários simulados Lynch and Park (2017).

Ao final, este trabalho apresentará o processo de desenvolvimento da aplicação, a integração entre o Unity 3D e o ROS 2 e a implementação dos algoritmos de *visual servoing*. Serão discutidos os desafios enfrentados, as soluções implementadas e os resultados obtidos, contribuindo para o avanço das soluções de controle visual em robótica e fornecendo uma ferramenta eficaz para pesquisa e ensino na área.

1.1 Trabalhos Relacionados

O uso de simuladores 3D em projetos de robótica tem sido amplamente explorado, com foco em ferramentas como V-Rep, Gazebo e Unity 3D de Melo et al. (2019). Esses simuladores são comparados em termos de usabilidade, liberdade de edição de cena, integração com o sistema ROS e interface do usuário. Unity 3D e V-Rep oferecem maior liberdade

[1] GitHub do ROS-TCP-Connector
[2] GitHub do ROS-TCP-Endpoint

e interfaces amigáveis, enquanto o Gazebo, embora limitado nesses aspectos, apresenta facilidade na integração com o ROS, sendo vantajoso para aplicações que exigem testes robustos em ambientes virtuais complexos antes da implementação física.

No contexto do controle visual, Li et al. (2017) abordam o *image-based visual servoing* (*IBVS*) para manipuladores de seis graus de liberdade, utilizando uma combinação de controle PD com controle em modo deslizante (SMC). A metodologia proposta aprimora a precisão e a estabilidade de sistemas de controle visual, destacando-se pela resistência a incertezas e rápida convergência, comprovada por simulações e experimentos.

Complementando essas abordagens, Kim et al. (2009) investigam o *eye-in-hand stereo visual servoing* em um braço robótico montado em uma cadeira de rodas para reconhecimento e manipulação de objetos em ambientes não estruturados. A estratégia de controle divide os movimentos em brutos e finos, onde o movimento bruto alinha o objeto no plano da imagem e o movimento fino ajusta-se para a manipulação precisa. A técnica demonstra robustez em condições com incertezas cinemáticas e calibração imperfeita, essencial para aplicações assistivas.

Em relação ao desenvolvimento de robôs em ambientes virtuais, Mattingly et al. (2012) propõem o uso do Unity 3D tanto em simulações de jogos quanto em robótica, destacando sua acessibilidade e flexibilidade para modelagem hierárquica e prototipagem rápida. A interface gráfica rica e o suporte a múltiplos tipos de *scripts* proporcionam uma alternativa prática para o desenvolvimento de robôs, especialmente em simulações detalhadas de interações robô-ambiente. Entretanto, Unity 3D apresenta limitações quando comparado a simuladores dedicados à robótica, como o Gazebo.

De acordo com Platt and Ricks (2022), as comparações entre ROS - Unity 3D e ROS - Gazebo evidenciam diferenças significativas que influenciam a escolha do simulador. Unity 3D se destaca pela escalabilidade, qualidade de sombras e suporte a ambientes virtuais de grande escala, sendo mais indicado para simulações que requerem alta fidelidade visual. Em contrapartida, Gazebo apresenta uma integração mais eficiente com o ROS e um uso otimizado de recursos computacionais, tornando-se mais adequado para simulações de menor escala e com requisitos mais modestos.

Mais recentemente, Hu et al. (2024) apresentam uma plataforma de ensino para simulação de um braço robótico de seis graus de liberdade, baseada no Unity 3D. O sistema integra a simulação visual com a operação do robô em cenários reais, facilitando o aprendizado de cinemática direta e inversa e permitindo ajustes nos parâmetros do ambiente virtual. Essa plataforma é vantajosa para o ensino prático, simulando com alta imersão e precisão o controle e a manipulação de um robô, embora careça de funcionalidades avançadas presentes em simuladores específicos para robótica.

Em síntese, os trabalhos revisados abordam diversas perspectivas de simulação e controle de robôs, destacando a importância de simuladores 3D com interfaces amigáveis e integração robusta com o ROS de Melo et al. (2019); Mattingly et al. (2012); Platt and

Ricks (2022), além de técnicas avançadas de controle visual para manipuladores Li et al. (2017); Kim et al. (2009). Inspirando-se nessas abordagens, o presente trabalho propõe um simulador em ambiente virtual que combina as vantagens destacadas nos estudos anteriores, focando em *visual servoing* e superando desafios como a escassez de robôs físicos para experimentação Hu et al. (2024) e a complexidade da integração de múltiplos sistemas Kim et al. (2009). Implementado na *engine* Unity 3D e integrado ao ROS 2 via ROS-TCP-Connector, similar às estratégias de Platt and Ricks (2022), o simulador facilita a troca de dados via *TCP/IP*, oferece portabilidade entre sistemas operacionais e proporciona um ambiente seguro para a validação de algoritmos de *visual servoing*. Dessa forma, contribui para o avanço das soluções de controle visual, promovendo inovação e aplicabilidade em cenários reais.

1.2 Motivação para o Projeto

A necessidade de uma plataforma de simulação robótica que integre alta fidelidade visual, facilidade de uso e eficiência computacional motiva este projeto. Estudos anteriores enfatizam a importância de interfaces amigáveis e integração robusta com o ROS de Melo et al. (2019); Mattingly et al. (2012); Platt and Ricks (2022), bem como a relevância de sistemas de controle visual precisos e adaptativos em ambientes não estruturados Li et al. (2017); Kim et al. (2009).

Considerando as vantagens do Unity 3D em qualidade visual e flexibilidade, apesar das limitações na integração com o ROS Mattingly et al. (2012); Platt and Ricks (2022), e a carência de funcionalidades avançadas em plataformas de ensino Hu et al. (2024), este projeto propõe o desenvolvimento de um simulador robótico em ambiente virtual. Implementado no Unity 3D e integrado ao ROS 2 via ROS-TCP-Connector, o simulador visa superar desafios como a escassez de robôs físicos e a complexidade de integração de sistemas múltiplos, focando em *visual servoing*.

Ao atender às necessidades identificadas nos estudos anteriores e avançar em soluções eficientes para o desenvolvimento e teste de algoritmos de *visual servoing*, o projeto contribui para o avanço das soluções de controle visual, promovendo inovação e aplicabilidade em cenários reais.

1.3 Objetivos

1.3.1 Objetivo Geral

Desenvolver uma aplicação que simule o comportamento de um robô controlado por *visual servoing* em um ambiente virtual, utilizando o Unity 3D e o ROS 2, permitindo a visualização, o controle e a análise do robô em tempo real, integrando a captura de imagens e

informações de sensores virtuais com algoritmos de controle visual.

1.3.2 Objetivos Específicos

Para atingir o objetivo geral, foram estabelecidos os seguintes objetivos específicos:

- Desenvolver um ambiente simulado no Unity 3D capaz de replicar cenários realistas, incluindo condições variáveis como iluminação, oclusões e sombras.
- Implementar a comunicação bidirecional entre o Unity 3D e o ROS 2 utilizando o pacote ROS-TCP-Connector, permitindo a troca de dados em tempo real.
- Implementar o algoritmo de *visual servoing* baseado em modelos teóricos estabelecidos, integrando-o ao sistema de simulação para o controle das juntas do robô.
- Realizar testes e validações da aplicação desenvolvida, avaliando a precisão e a eficiência do controle visual em diferentes cenários simulados.
- Documentar todo o processo de desenvolvimento, apresentando os desafios enfrentados e as soluções implementadas.

1.4 Estrutura do Texto

Este trabalho está organizado em cinco capítulos. No Capítulo 1, apresenta-se a introdução ao tema, destacando a relevância da pesquisa, os principais conceitos envolvidos e os objetivos do trabalho. O Capítulo 2 trata da fundamentação teórica, abordando os conceitos essenciais de *visual servoing*, simulação em robótica, Unity 3D e ROS 2, além de uma revisão da literatura relacionada. No Capítulo 3, detalha-se o processo de desenvolvimento da aplicação, incluindo a configuração do ambiente de simulação, a integração entre os sistemas e a implementação dos algoritmos de controle visual. O Capítulo 4 apresenta os resultados obtidos com a aplicação desenvolvida, discutindo os testes realizados e a validação dos algoritmos. Por fim, o Capítulo 5 oferece as considerações finais, discutindo as contribuições do trabalho, suas limitações e sugestões para pesquisas futuras.

Capítulo 2

Fundamentação Teórica

Esta fundamentação teórica explora os três pilares fundamentais que sustentam o desenvolvimento deste estudo: cinemática de robôs, *visual servoing* e computação gráfica. Inicialmente, aborda-se a cinemática de robôs, que tem por objetivo permitir a manipulação e o controle dos movimentos do manipulador. Em seguida, o *visual servoing* é discutido, destacando a utilização de informações visuais para guiar a atuação dos robôs de maneira adaptativa e responsiva ao ambiente. Por último, a computação gráfica é apresentada como uma ferramenta para a visualização e simulação avançada dos sistemas robóticos. A inter-relação entre esses temas permite a integração de manipulação precisa (cinemática) com *feedback* visual (*visual servoing*) e suporte computacional robusto (computação gráfica). Essa sinergia propicia o desenvolvimento de sistemas robóticos mais sofisticados, eficientes e adaptáveis às variadas exigências do ambiente operacional.

2.1 Cinemática de Robôs

A cinemática de robôs é a base para descrever os movimentos de manipuladores robóticos. Envolve o estudo da relação entre as coordenadas das juntas do robô e a posição e orientação do seu efetuador final, sem considerar as forças que atuam no sistema para a realização do movimento. Os conceitos abordados incluem cinemática direta, inversa e diferencial, além do tratamento de singularidades e pseudo-inversas, com o objetivo de evitar configurações problemáticas em robôs com redundância.

2.1.1 Cinemática Direta, Inversa e Diferencial

Cinemática Direta

A cinemática direta determina a posição e orientação do efetuador final a partir dos ângulos das juntas. Esta é uma das primeiras etapas no planejamento de movimentos em robótica e é amplamente abordada por Craig (2016); Lynch and Park (2017), com métodos baseados em transformações homogêneas.

Matematicamente, a cinemática direta pode ser expressa utilizando as matrizes de transformação homogênea. Para um robô com n juntas, a posição e orientação do efetuador final T é dada por:

$$T = A_1 A_2 \dots A_n \tag{2.1}$$

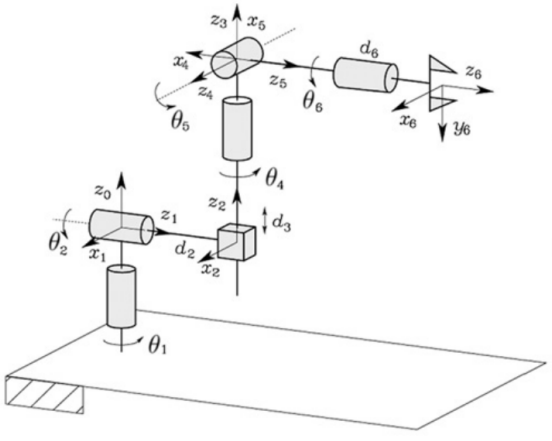
onde cada A_i é a matriz de transformação homogênea que descreve a relação entre a junta i e a junta $i+1$, definida pelos parâmetros de Denavit-Hartenberg:

$$A_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

onde:

- θ_i é o ângulo da junta,
- α_i é o ângulo entre os eixos z_i e z_{i+1},
- a_i é o deslocamento ao longo de x_i,
- d_i é o deslocamento ao longo de z_i.

A Figura 2.1 ilustra um exemplo dos parâmetros de Denavit-Hartenberg aplicados ao manipulador Stanford. Essa representação gráfica auxilia na visualização da relação entre os parâmetros e a geometria do robô.



D-H parameters

Link	a_i	α_1	d_i	θ_i
1	0	$-\pi/2$	0	θ_1
2	0	$\pi/2$	d_2	θ_2
3	0	0	d_3	0
4	0	$-\pi/2$	0	θ_4
5	0	$\pi/2$	0	θ_5
6	0	0	d_6	θ_6

Figura 2.1: Manipulador Stanford e seus parâmetros de Denavit-Hartenberg.

Além da fundamentação teórica, a implementação computacional da cinemática direta permite o planejamento e controle de robôs. A programação desses algoritmos geralmente é realizada em linguagens como C ou Python, que oferecem bibliotecas poderosas para

cálculo numérico e simulação. Um exemplo básico de implementação da cinemática direta utilizando a biblioteca NumPy pode ser representado pela seguinte função:

```
import numpy as np

def forward_kinematics(theta, dh_params):
    T = np.eye(4)
    for i in range(len(theta)):
        a, alpha, d, theta_i = dh_params[i]
        T_i = np.array(
            [
                [
                    np.cos(theta_i),
                    -np.sin(theta_i) * np.cos(alpha),
                    np.sin(theta_i) * np.sin(alpha),
                    a * np.cos(theta_i),
                ],
                [
                    np.sin(theta_i),
                    np.cos(theta_i) * np.cos(alpha),
                    -np.cos(theta_i) * np.sin(alpha),
                    a * np.sin(theta_i),
                ],
                [0, np.sin(alpha), np.cos(alpha), d],
                [0, 0, 0, 1],
            ]
        )
        T = np.dot(T, T_i)
    return T
```

Código 2.1: Implementação da cinemática direta em Python

Este exemplo mostra a multiplicação das matrizes de transformação homogênea para calcular a posição e orientação do efetuador final a partir dos ângulos das juntas θ e dos parâmetros de Denavit-Hartenberg.

Cinemática Inversa

A cinemática inversa trata do problema de determinar as configurações articulares necessárias para que o efetuador final de um robô atinja uma posição e orientação desejadas no espaço de trabalho. Diferentemente da cinemática direta, que realiza o mapeamento dos ângulos das juntas para posições no espaço cartesiano, a cinemática inversa busca os valores dos ângulos articulares a partir de uma posição e orientação especificadas. Este problema é intrinsecamente mais complexo, devido à sua natureza não linear e frequentemente multivalorada, como discutido em Siciliano et al. (2009); Spong et al. (2020).

Matematicamente, a cinemática inversa pode ser representada como a solução do

seguinte problema:

$$\boldsymbol{\theta} = f^{-1}(\mathbf{x}_d), \tag{2.3}$$

onde:

- $\mathbf{x}_d \in SE(3)$ é o vetor que representa a posição e orientação desejadas do efetuador final no espaço de trabalho;
- $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots, \theta_n]^T$ é o vetor de ângulos articulares a ser determinado;
- $SE(3)$ é o grupo especial euclidiano que define as transformações rígidas no espaço tridimensional.

Em muitos casos, o problema da cinemática inversa é reformulado como um problema de otimização para minimizar a diferença entre a posição e orientação desejadas ($\mathbf{x}_d$) e a posição e orientação calculadas pela cinemática direta ($f(\boldsymbol{\theta})$):

$$\min_{\boldsymbol{\theta}} \|\mathbf{x}_d - f(\boldsymbol{\theta})\|, \tag{2.4}$$

sujeito às restrições:

$$\theta_{\min} \leq \theta_i \leq \theta_{\max}, \quad \forall i \in \{1, 2, \ldots, n\}, \tag{2.5}$$

onde $\theta_{\min}$ e $\theta_{\max}$ são os limites inferiores e superiores, respectivamente, de cada junta do robô.

Como soluções analíticas geralmente não estão disponíveis para manipuladores com configurações geométricas complexas, métodos numéricos e algoritmos de otimização são amplamente utilizados. Técnicas baseadas na Jacobiana são comuns para ajustar iterativamente $\boldsymbol{\theta}$ de forma que $f(\boldsymbol{\theta})$ se aproxime de $\mathbf{x}_d$. Em casos em que múltiplas soluções são possíveis, critérios secundários, como evitar singularidades, minimizar energia ou prevenir colisões, podem ser adicionados ao modelo.

Cinemática Diferencial

A cinemática diferencial estuda a relação entre as velocidades articulares de um manipulador robótico e as velocidades linear e angular do efetuador final no espaço operacional. Essa relação é descrita matematicamente pela matriz Jacobiana, que é derivada a partir das equações de cinemática direta. A matriz Jacobiana permite transformar as velocidades articulares em velocidades no espaço operacional, sendo usada para a análise e controle de robôs manipuladores Siciliano et al. (2009).

Para um manipulador robótico, a matriz Jacobiana $\mathbf{J}$ relaciona o vetor de velocidades articulares $\dot{\mathbf{q}}$ ao vetor que combina a velocidade linear $\mathbf{v}$ e a velocidade angular ω do efetuador final, conforme a seguinte expressão:

$$\begin{bmatrix} \mathbf{v} \\ \omega \end{bmatrix} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}. \tag{2.6}$$

onde:

- $\mathbf{v}$ é o vetor velocidade linear do efetuador final no espaço operacional,
- ω é o vetor velocidade angular do efetuador final no espaço operacional,
- $\mathbf{q}$ é o vetor de coordenadas articulares do robô,
- $\dot{\mathbf{q}}$ representa o vetor de velocidades articulares,
- $\mathbf{J}(\mathbf{q})$ é a matriz Jacobiana, que depende das configurações articulares $\mathbf{q}$.

A Jacobiana também permite identificar configurações críticas, conhecidas como singularidades, nas quais o manipulador perde graus de liberdade efetivos. Esse tema será explorado na próxima seção, junto com abordagens robustas para lidar com essas condições. Adicionalmente, a análise da Jacobiana será usada no desenvolvimento de estratégias de controle das seções seguintes.

2.1.2 Singularidades e Pseudo-Inversa

As singularidades representam configurações críticas em que o manipulador robótico perde a capacidade de se mover em certas direções ou aplicar forças específicas. Matematicamente, essas situações ocorrem quando a matriz Jacobiana $\mathbf{J}$ perde posto completo, ou seja, quando:

$$\det(\mathbf{J}\mathbf{J}^T) = 0. \tag{2.7}$$

Nessas configurações, o robô pode apresentar instabilidade ou movimentos indesejados, prejudicando o desempenho. Para lidar com esses problemas, utiliza-se a pseudo-inversa de Moore-Penrose, definida por:

$$\mathbf{J}^+ = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}. \tag{2.8}$$

A pseudo-inversa permite encontrar soluções mínimas para $\dot{\mathbf{q}}$, garantindo que o erro em $\dot{\mathbf{x}}$ seja minimizado. Essa relação pode ser expressa como:

$$\dot{\mathbf{q}} = \mathbf{J}^+\dot{\mathbf{x}}, \tag{2.9}$$

onde $\dot{\mathbf{x}}$ é o vetor de velocidades desejadas no espaço operacional. O uso da pseudo-inversa é uma técnica fundamental para lidar com os desafios relacionados a movimentos

e forças em manipuladores robóticos, especialmente em situações onde a singularidade pode comprometer o controle preciso do sistema Corke (2023).

No entanto, a utilização da pseudo-inversa de Moore-Penrose pode levar a valores muito grandes de $\dot{\mathbf{q}}$ próximo às singularidades, devido à inversão de valores próximos de zero na matriz $\mathbf{J}\mathbf{J}^T$. Para mitigar esse problema, emprega-se a pseudo-inversa amortecida, também conhecida como método dos mínimos quadrados com amortecimento (DLS - Damped Least Squares), definida por:

$$\mathbf{J}_\lambda^+ = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \lambda^2\mathbf{I})^{-1}, \tag{2.10}$$

onde λ é um fator de amortecimento escalar positivo e $\mathbf{I}$ é a matriz identidade.

A inclusão do termo $\lambda^2\mathbf{I}$ evita a inversão de matrizes singulares ou quase singulares, estabilizando o cálculo de $\dot{\mathbf{q}}$ mesmo em proximidade de singularidades. A relação resultante é então:

$$\dot{\mathbf{q}} = \mathbf{J}_\lambda^+\dot{\mathbf{x}}. \tag{2.11}$$

O valor de λ pode ser ajustado de acordo com a proximidade à singularidade; valores maiores de λ aumentam o amortecimento, reduzindo oscilações e movimentos abruptos.

Assim, a pseudo-inversa amortecida oferece uma solução mais estável e robusta para o controle de manipuladores em regiões próximas a singularidades, melhorando o desempenho operacional e a segurança do sistema robótico.

2.1.3 Controle de Juntas

O controle das juntas é usado para garantir que o manipulador execute movimentos precisos, suaves e estáveis. Os controladores são projetados para regular as velocidades ou posições das juntas, dependendo das especificações da tarefa e do desempenho desejado. Neste trabalho, foi adotada a estratégia de controladores de velocidade, uma vez que a cinemática diferencial, através da matriz Jacobiana, mapeia diretamente as velocidades no espaço operacional para as velocidades articulares, simplificando a implementação e atendendo aos objetivos das simulações realizadas Siciliano et al. (2009).

Controladores de Velocidade

Controladores de velocidade regulam diretamente a taxa de variação das juntas, sendo amplamente aplicados em tarefas que exigem rápidas adaptações às condições do espaço operacional. A lei de controle, propocional ao erro, pode ser expressa como:

$$\dot{\mathbf{q}} = K_v(\mathbf{q}_d - \mathbf{q}), \tag{2.12}$$

onde:

- K_v é o ganho proporcional de velocidade,
- $\mathbf{q}_d$ é o vetor de velocidades desejadas das juntas,
- $\mathbf{q}$ representa as velocidades articulares atuais.

Esse tipo de controlador destaca-se por sua simplicidade de implementação e eficiência em responder a alterações rápidas no ambiente. Além disso, os controladores de velocidade oferecem maior suavidade nos movimentos e ajudam a minimizar erros de trajetória.

2.1.4 Programando o Comportamento de Robôs

A programação do comportamento de robôs é fundamental para garantir que estes possam executar tarefas de forma autônoma e eficiente. Uma abordagem comum para gerenciar os diferentes estados e transições do comportamento robótico é a utilização de máquinas de estado finito (FSM).

Máquinas de Estado Finito (FSM)

As máquinas de estado finito (FSM) são modelos computacionais que descrevem sistemas com base em estados finitos e nas transições entre eles, que ocorrem em resposta a eventos ou condições específicas. Na robótica, esses modelos são amplamente utilizados para controlar sequências de ações, reagir a estímulos do ambiente e gerenciar diferentes modos de operação de um robô Balogh and Obdržálek (2019).

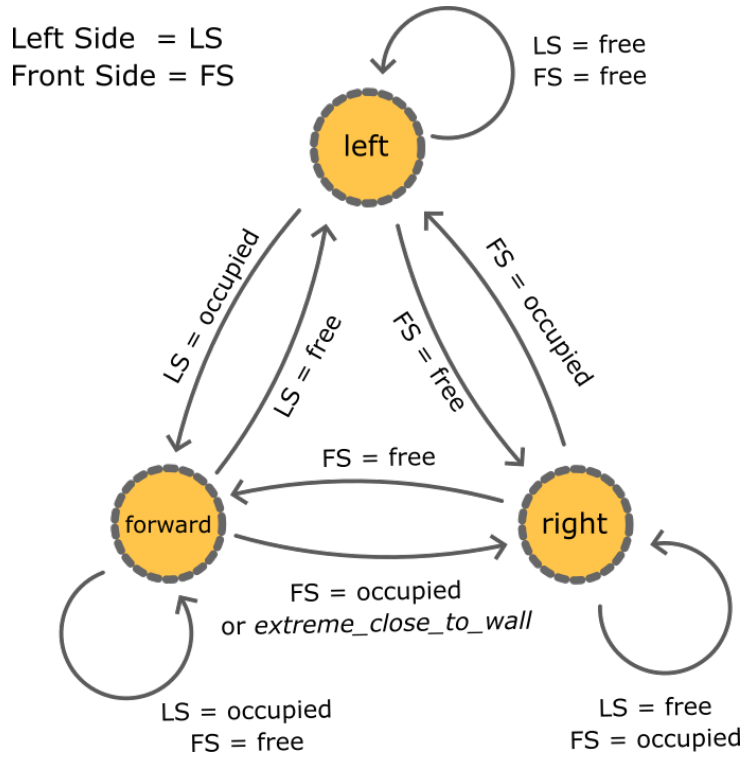


Figura 2.2: Diagrama de uma máquina de estados finita.

No exemplo ilustrado pela Figura 2.2, o robô opera em três estados distintos: ”left”, ”forward”e ”right”, representando as ações de virar à esquerda, mover para frente e virar à direita, respectivamente. As transições entre esses estados são acionadas por eventos ou condições detectadas durante a operação, e cada estado possui regras que definem como as mudanças ocorrem. Além disso, é possível retornar ao mesmo estado em casos específicos, como indicado pelas setas de laço.

Esse tipo de FSM pode ser aplicado em diversos contextos, como navegação autônoma, onde o robô alterna entre estados para ajustar sua trajetória com base em sensores ou comandos externos. O modelo facilita a organização lógica e o gerenciamento do comportamento do robô, permitindo respostas rápidas e consistentes a mudanças no ambiente.

Implementação de FSM em Python

A implementação de uma FSM pode ser realizada utilizando classes que representam os diferentes estados e gerenciam as transições. A seguir, apresenta-se um exemplo básico de uma FSM em Python:

```
class State:
    def __init__(self, name):
        self.name = name

    def on_event(self, event):
        pass

class IdleState(State):
    def __init__(self):
        super().__init__('Idle')

    def on_event(self, event):
        if event == 'start_moving':
            return MovingState()
        return self

class MovingState(State):
    def __init__(self):
        super().__init__('Moving')

    def on_event(self, event):
        if event == 'obstacle_detected':
            return ObstacleState()
        elif event == 'stop':
            return IdleState()
        return self

class ObstacleState(State):
```

```
29      def __init__(self):
30          super().__init__('Obstacle')
31
32      def on_event(self, event):
33          if event == 'clear_path':
34              return MovingState()
35          elif event == 'stop':
36              return IdleState()
37          return self
38
39  class FSM:
40      def __init__(self):
41          self.state = IdleState()
42
43      def on_event(self, event):
44          self.state = self.state.on_event(event)
45          print(f"Transição para o estado: {self.state.name}")
46
47  fsm = FSM()
48  fsm.on_event('start_moving')
49  fsm.on_event('obstacle_detected')
50  fsm.on_event('clear_path')
51  fsm.on_event('stop')
```

Código 2.2: Implementação de uma Máquina de Estado Finito em Python

Neste exemplo, cada estado é representado por uma classe que herda da classe base `State`. A classe `FSM` gerencia o estado atual e atualiza o estado com base nos eventos recebidos. Ao chamar o método `on_event`, a FSM transita para o próximo estado apropriado e imprime a transição realizada. Dessa forma, a implementação modular e orientada a objetos facilita a manutenção e expansão do comportamento do robô, demonstrando a utilização das Máquinas de Estado Finito na programação de sistemas robóticos complexos.

2.2 *Visual Servoing*

O *visual servoing* refere-se ao uso de dados visuais para controlar o movimento de robôs em tempo real. Este conceito integra câmeras como sensores primários para guiar robôs de forma adaptativa, permitindo que o robô ajuste seus movimentos com base nas informações visuais recebidas do ambiente.

2.2.1 Modelagem da Câmera

A modelagem da câmera em sistemas de *visual servoing* é essencial para compreender a transformação de coordenadas no espaço tridimensional para o plano bidimensional da

imagem. Este processo é baseado na geometria projetiva e inclui conceitos como formação de imagem, plano discreto e matriz da câmera.

Formação de Imagem

A formação de imagem em câmeras perspécticas é descrita pela projeção de pontos 3D no plano da imagem, como mostrado na Equação:

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}, \tag{2.13}$$

onde f é a distância focal da câmera. A projeção é normalizada dividindo-se as coordenadas por $\tilde{z}$:

$$x = \frac{\tilde{x}}{\tilde{z}}, \quad y = \frac{\tilde{y}}{\tilde{z}}. \tag{2.14}$$

Isso resulta nas coordenadas x, y do plano da imagem, referidas como coordenadas normalizadas ou plano retiniano.

Plano de Imagem Discreto

No plano de imagem discreto, os pixels são organizados em uma grade $W \times H$, com origem no canto superior esquerdo. As coordenadas dos pixels são denotadas como u, v e relacionadas às coordenadas normalizadas x, y pela matriz de calibração intrínseca da câmera:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \tag{2.15}$$

onde f_x, f_y são as distâncias focais em termos de pixels e c_x, c_y representam o centro ótico.

Matriz da Câmera

A matriz da câmera combina os parâmetros intrínsecos e extrínsecos (posição e orientação da câmera no espaço). O modelo geral de projeção pode ser expresso como:

$$\mathbf{p} = \mathbf{K}[\mathbf{R}|\mathbf{t}]\mathbf{P}, \tag{2.16}$$

onde:

- $\mathbf{K}$: matriz intrínseca;

- $[\mathbf{R}|\mathbf{t}]$: rotação ($\mathbf{R}$) e translação ($\mathbf{t}$) da câmera;
- $\mathbf{P}$: coordenada homogênea do ponto no espaço 3D.

Essa matriz é fundamental para modelar como os pontos 3D são projetados no plano da imagem, fornecendo a base para o *image-based visual servoing* (*IBVS*), uma abordagem em que o controle do robô é realizado diretamente a partir das características extraídas do plano da imagem. Alternativamente, existem as abordagens baseadas em posição (*position-based visual servoing*, PBVS) e híbridas (*hybrid visual servoing*, HBVS), que também utilizam informações visuais para o controle, mas de maneiras diferentes.

2.2.2 *Image-based Visual Servoing* (*IBVS*)

No *IBVS*, o controle do robô é realizado diretamente com base nas características extraídas da imagem. Este método evita a reconstrução explícita do espaço tridimensional, simplificando o processo. A relação fundamental utilizada é entre as velocidades no plano da imagem $(\dot{u}, \dot{v})$ e as velocidades da câmera $(v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$:

$$\begin{bmatrix} \dot{u}_i \\ \dot{v}_i \end{bmatrix} = J_p(u_i, v_i, Z_i) \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \tag{2.17}$$

onde $J_p(u, v, Z)$ é a matriz Jacobiana da imagem, responsável por relacionar as velocidades no espaço da câmera com os movimentos percebidos no plano da imagem. Para múltiplos pontos na imagem, a relação geral é dada por:

$$\begin{bmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \end{bmatrix} = \begin{bmatrix} J_p(u_1, v_1, Z_1) \\ J_p(u_2, v_2, Z_2) \\ J_p(u_3, v_3, Z_3) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}. \tag{2.18}$$

A matriz Jacobiana da imagem $J_p(u, v, Z)$ é explicitamente definida como:

$$J_p(u, v, Z) = \begin{bmatrix} -\hat{f}/Z & 0 & u/Z & uv/\hat{f} & -(\hat{f} + u^2)/\hat{f} & v \\ 0 & -\hat{f}/Z & v/Z & (\hat{f} + v^2)/\hat{f} & -uv/\hat{f} & -u \end{bmatrix}, \tag{2.19}$$

onde:

- $\hat{f}$: distância focal da câmera,
- Z: profundidade do ponto no espaço da câmera,
- (u, v): coordenadas do ponto no plano da imagem.

Essa matriz relaciona as velocidades lineares e angulares da câmera às velocidades percebidas dos pixels no plano da imagem. O objetivo do *IBVS* é minimizar o erro entre as posições atuais dos pixels (u_i, v_i) e as posições desejadas (u_i^*, v_i^*). Para isso, a velocidade desejada dos pixels é definida como:

$$\begin{bmatrix} \dot{u}_i^* \\ \dot{v}_i^* \end{bmatrix} = \lambda \begin{bmatrix} u_i^* - u_i \\ v_i^* - v_i \end{bmatrix}, \tag{2.20}$$

onde:

- $\lambda > 0$: ganho de controle proporcional,
- (u_i^*, v_i^*): coordenadas desejadas dos pixels,
- (u_i, v_i): coordenadas atuais dos pixels.

A velocidade da câmera necessária para corrigir o erro visual pode então ser calculada resolvendo a seguinte equação:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \lambda \begin{bmatrix} J_p(u_1, v_1, Z_1) \\ J_p(u_2, v_2, Z_2) \\ J_p(u_3, v_3, Z_3) \end{bmatrix}^{-1} \begin{bmatrix} p_1^* - p_1 \\ p_2^* - p_2 \\ p_3^* - p_3 \end{bmatrix}, \tag{2.21}$$

onde $p_i = [u_i, v_i]^T$ são as coordenadas dos pixels no plano da imagem, e $p_i^* = [u_i^*, v_i^*]^T$ são as coordenadas desejadas. Essa formulação incorpora a Jacobiana projetada J_p, os ganhos de controle, e o erro visual $(p_i^* - p_i)$ para calcular diretamente as velocidades da câmera necessárias para alinhar os pontos da imagem com suas posições desejadas. Este controle ocorre em tempo real, possibilitando tarefas como alinhamento de câmera, rastreamento de objetos e manipulação visual.

2.3 Computação Gráfica

A computação gráfica é uma disciplina central na criação e manipulação de imagens e modelos tridimensionais em ambientes virtuais. Ela abrange uma variedade de técnicas e algoritmos que permitem a representação visual de objetos e cenários complexos. Nesta

seção, são explorados os conceitos teóricos fundamentais que sustentam a modelagem 3D, mapeamento UV, texturização, renderização, utilização de shaders e pipelines de renderização, todos essenciais para a construção de ambientes virtuais sofisticados e realistas.

2.3.1 Modelagem 3D

A modelagem tridimensional é o processo de criar representações digitais de objetos no espaço tridimensional. Este processo envolve a definição da geometria, topologia e estrutura dos modelos, permitindo a criação de formas complexas e detalhadas Hughes (2014). As principais técnicas de modelagem 3D incluem:

- **Modelagem Poligonal**: Utiliza polígonos, principalmente triângulos ou quadriláteros, para representar a superfície de objetos 3D. A qualidade e o nível de detalhamento do modelo são determinados pela densidade e organização dos polígonos.
- **NURBS (Non-Uniform Rational B-Splines)**: Emprega curvas e superfícies paramétricas para criar superfícies suaves, oferecendo maior precisão no controle da curvatura e suavidade dos modelos Piegl and Tiller (1996).
- **Escultura Digital**: Utiliza ferramentas de escultura virtual que simulam o processo de escultura tradicional, permitindo uma modelagem mais intuitiva e artística[1].

No processo de modelagem 3D, é comum criar inicialmente modelos highpoly, que possuem alta contagem de polígonos e detalhes precisos. No entanto, para otimizar o desempenho em aplicações em tempo real, utiliza-se a retopologia como uma etapa fundamental do fluxo de trabalho.

A retopologia é uma técnica especializada que visa otimizar a malha (mesh) de um modelo, convertendo modelos highpoly em lowpoly através da redução controlada da contagem de polígonos. Esta otimização é crucial para melhorar a eficiência e adaptabilidade em aplicações que exigem processamento em tempo real, como simulações e jogos[2]. O processo envolve a reorganização sistemática da topologia da malha, garantindo uma distribuição uniforme dos polígonos que facilita tanto a aplicação de animações quanto a renderização eficiente.

Durante a retopologia, o desafio principal está em preservar a forma e os detalhes essenciais do modelo original enquanto se implementa uma estrutura de polígonos mais eficiente. Este processo requer um equilíbrio meticuloso entre a manutenção da fidelidade visual e a otimização do desempenho, especialmente nas áreas do modelo que necessitarão de animação ou deformação durante sua utilização.

[1] Guia sobre escultura do Zbrush
[2] Documentação sobre retopologia do Blender

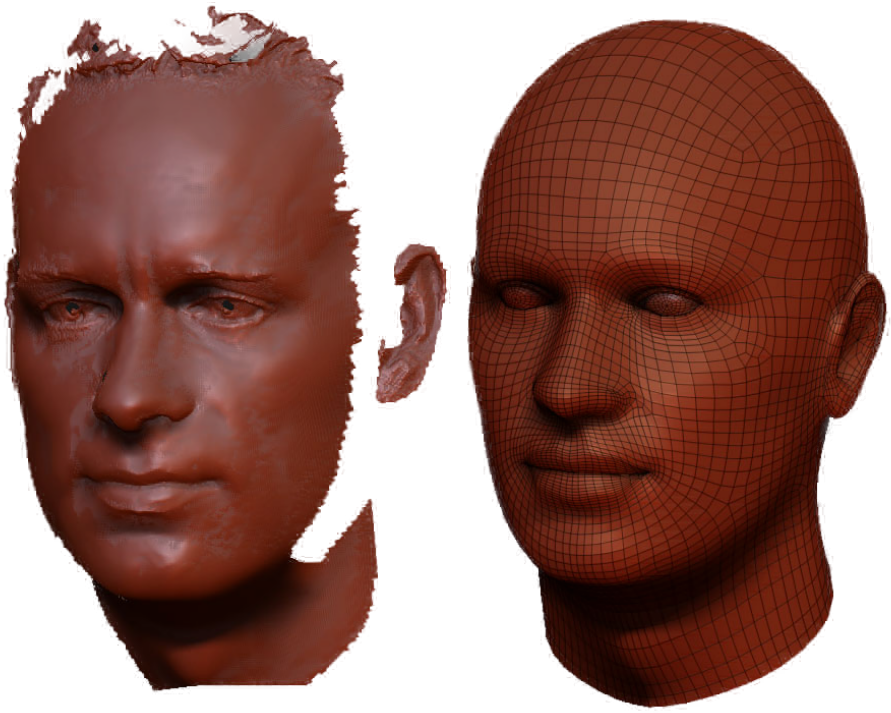


Figura 2.3: Exemplo de retopologia aplicada a um modelo 3D.

A Figura 2.3 demonstra um exemplo prático de retopologia, onde é possível observar a transformação de uma malha original com alta densidade de polígonos e topologia irregular para uma malha retopologizada que apresenta uma estrutura mais organizada e otimizada para renderização em tempo real.

2.3.2 Mapeamento UV

O mapeamento UV é uma técnica fundamental na computação gráfica que consiste em projetar a superfície tridimensional de um modelo em um plano bidimensional Flavell (2010). Este processo é usado na aplicação de texturas sobre modelos 3D, permitindo que imagens 2D sejam corretamente mapeadas e alinhadas com a geometria do objeto [3]. O mapeamento UV envolve a criação de coordenadas UV, onde cada ponto na superfície do modelo recebe uma correspondência em um espaço 2D, garantindo que as texturas sejam aplicadas sem distorções ou sobreposições indesejadas.

Matematicamente, o mapeamento UV pode ser representado por uma função $f : S \to \mathbb{R}^2$, onde S é a superfície do modelo 3D e $\mathbb{R}^2$ representa o espaço bidimensional da textura. A função f define a correspondência entre cada vértice ou face do modelo 3D e suas coordenadas na textura 2D.

$$f : (x, y, z) \mapsto (u, v) \tag{2.22}$$

[3]Documentação sobre mapeamento UV do Autodesk Maya

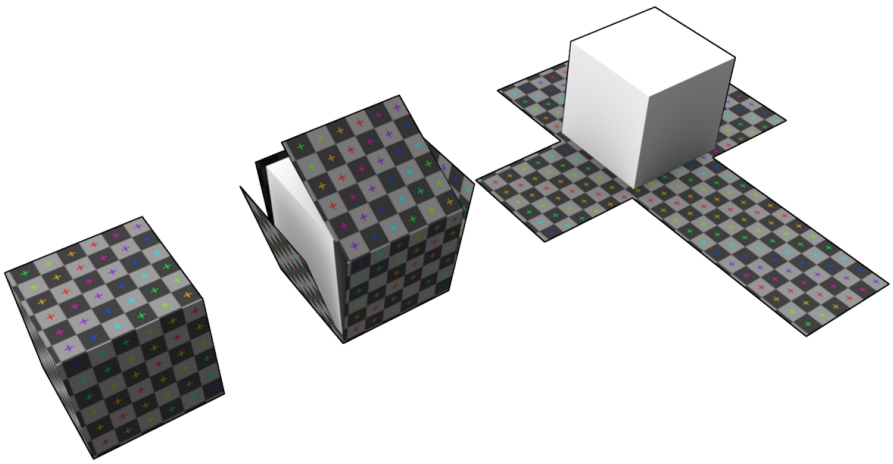


Figura 2.4: Exemplo de mapeamento UV aplicado a um modelo 3D.

A Figura 2.4 ilustra o processo de mapeamento UV aplicado a um cubo. Neste procedimento, a malha tridimensional do cubo é desdobrada em um plano bidimensional, permitindo que uma textura 2D seja aplicada de forma precisa e sem distorções significativas. Cada face do cubo é associada a uma coordenada UV específica, garantindo que a imagem da textura seja corretamente alinhada e mapeada sobre a superfície 3D. Assim, a malha achatada facilita a aplicação uniforme das texturas, aprimorando a fidelidade visual do objeto tridimensional Flavell (2010).

2.3.3 Texturização

A texturização refere-se à aplicação de imagens (*textures*) sobre a superfície de modelos 3D para simular propriedades visuais como cor, reflexividade, rugosidade e outros atributos materiais. Este processo enriquece a aparência dos modelos, conferindo-lhes um realismo maior sem a necessidade de aumentar a complexidade geométrica.

As técnicas avançadas de texturização incluem, mas não se limitam a:

- **Mapas Normais (Normal Maps)**: Utilizados para simular detalhes de superfície sem aumentar a contagem de polígonos. Eles modificam as normais das superfícies durante a renderização para criar a ilusão de relevo, permitindo que pequenas variações na superfície sejam percebidas sem a necessidade de geometria adicional.
- **Mapas de Deslocamento (Displacement Maps)**: Alteram a geometria do modelo em tempo real com base na textura, permitindo a criação de detalhes complexos

e superfícies irregulares. Diferentemente dos mapas normais, os mapas de deslocamento realmente modificam a posição dos vértices, proporcionando um nível mais alto de detalhe.

- **Mapas de Ambient Occlusion (AO Maps)**: Simulam sombras suaves em áreas onde a luz ambiental é reduzida, aumentando a profundidade visual e a percepção de detalhes. Esses mapas ajudam a destacar a topologia do modelo, melhorando a definição das áreas de contato e recessos.

A aplicação adequada desses mapas permite a criação de materiais realistas que interagem de forma convincente com as fontes de luz no ambiente virtual. Além dos mapas mencionados, outras técnicas como specular maps, metalness maps e emission maps também são amplamente utilizadas para aprimorar ainda mais a fidelidade visual dos modelos 3D.

2.3.4 Renderização

A renderização é o processo de converter um modelo 3D em uma imagem 2D, integrando cálculos de iluminação, sombras, reflexões e outros efeitos visuais para criar uma representação visual da cena. Diferentes técnicas de renderização são empregadas, variando em complexidade, desempenho e qualidade visual Hearn et al. (2011).

Rasterização

A rasterização é a técnica predominante para renderização em tempo real, especialmente em aplicações como jogos e simulações interativas. O processo envolve a projeção de vértices de polígonos no espaço tridimensional para a tela bidimensional, transformando-os em pixels. Em seguida, são aplicados texturas e cálculos de iluminação para gerar a aparência final de cada objeto.

Essa abordagem é altamente eficiente devido à sua capacidade de processar grandes quantidades de dados em paralelo, aproveitando arquiteturas de hardware como GPUs (Unidades de Processamento Gráfico). Além disso, a rasterização suporta técnicas avançadas, como mapeamento de sombras (*shadow mapping*) e efeitos baseados em telas (*screen-space effects*), que melhoram a qualidade visual sem comprometer significativamente o desempenho Akenine-Möller et al. (2018).

Ray Tracing

O *Ray Tracing* é uma técnica de renderização que simula o comportamento físico da luz para gerar imagens com alto nível de realismo. Em vez de apenas processar polígonos visíveis, como na rasterização, o *Ray Tracing* emite raios a partir da câmera, rastreando

suas interações com os objetos da cena. Essas interações incluem reflexões, refrações, sombras e dispersão da luz.

Esse método permite a criação de efeitos de iluminação mais precisos, como sombras suaves, reflexões complexas e iluminação global, que são difíceis de alcançar com técnicas baseadas em rasterização. A Figura 2.5 ilustra o fluxo básico do *Ray Tracing*, destacando como os raios emitidos pela câmera interagem com as superfícies na cena.

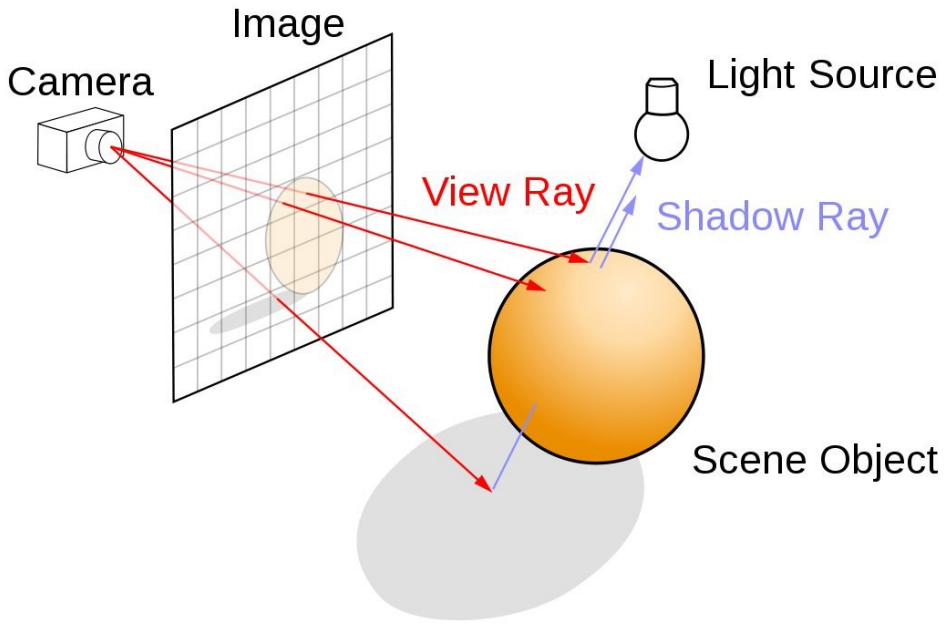


Figura 2.5: Diagrama ilustrativo do processo de *Ray Tracing* em uma cena 3D.

Embora o *Ray Tracing* tenha historicamente sido restrito a renderizações offline devido à sua alta demanda computacional, avanços recentes em hardware e software têm possibilitado sua utilização em tempo real. Conforme discutido por Yu (2022), técnicas híbridas, que combinam abordagens de *Ray Tracing* direto e reverso, têm emergido como soluções viáveis para equilibrar eficiência e precisão em aplicações interativas. Adicionalmente, GPUs modernas otimizadas para cálculos paralelos e APIs específicas, como NVIDIA RTX e Microsoft DirectX Raytracing (DXR), têm desempenhado um papel crucial na popularização dessa técnica para jogos e outros cenários interativos.

Técnicas híbridas, que combinam rasterização e *Ray Tracing*, também vêm ganhando popularidade. Essas abordagens aproveitam a eficiência da rasterização para renderizar a geometria base, enquanto utilizam o *Ray Tracing* para efeitos específicos, como reflexos e sombras, equilibrando qualidade visual e desempenho computacional.

2.3.5 Shaders

Shaders são programas de computador que determinam como os objetos são renderizados na tela, controlando aspectos como iluminação, cor, textura e efeitos visuais especiais

Rost et al. (2009). Eles são executados na GPU (Unidade de Processamento Gráfico) e são essenciais para personalizar o pipeline de renderização.

Tipos de Shaders

- **Vertex Shaders**: Processam cada vértice individualmente, transformando suas coordenadas de modelo para coordenadas de tela e aplicando efeitos como deformações e animações.
- **Fragment Shaders**: Calculam a cor de cada fragmento (pixel potencial) na tela, aplicando texturas, iluminação e outros efeitos visuais.
- **Geometry Shaders**: Operam sobre primitivas geométricas (como triângulos) e podem gerar novas primitivas ou modificar as existentes.
- **Compute Shaders**: Utilizados para cálculos gerais não necessariamente ligados ao pipeline de renderização, permitindo operações paralelas complexas.

Programação de Shaders

Os shaders são escritos em linguagens específicas, como GLSL (OpenGL Shading Language) ou HLSL (High-Level Shading Language) para DirectX. Um exemplo básico de um *vertex shader* em GLSL é apresentado a seguir:

```
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;
    TexCoords = aTexCoords;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

Código 2.3: Exemplo de Vertex Shader em GLSL

O código acima demonstra a transformação das coordenadas de vértice e o cálculo das normais no espaço do mundo, preparando os dados para o *fragment shader*.

2.3.6 Pipeline de Renderização

O pipeline de renderização é a sequência de etapas pelas quais os dados gráficos são processados para gerar a imagem final. Ele inclui transformação de vértices, montagem de primitivas, rasterização, execução de *shaders* e processamento de fragmentos Akenine-Möller et al. (2018).

Etapas do Pipeline

1. **Transformação de Vértices**: Os vértices são transformados do espaço do modelo para o espaço do mundo, depois para o espaço da câmera e finalmente para o espaço de projeção.

2. **Montagem de Primitivas**: Os vértices transformados são agrupados em primitivas geométricas, como triângulos.

3. **Rasterização**: As primitivas são convertidas em fragmentos (pixels potenciais) que serão processados para determinar sua cor final.

4. **Execução de Shaders**: Shaders de fragmentos calculam a cor de cada fragmento, aplicando texturas, iluminação e outros efeitos.

5. **Teste de Profundidade e Blend**: Os fragmentos são testados quanto à profundidade para determinar se devem ser visíveis ou ocultos por outros objetos, e então são combinados com os pixels existentes na tela.

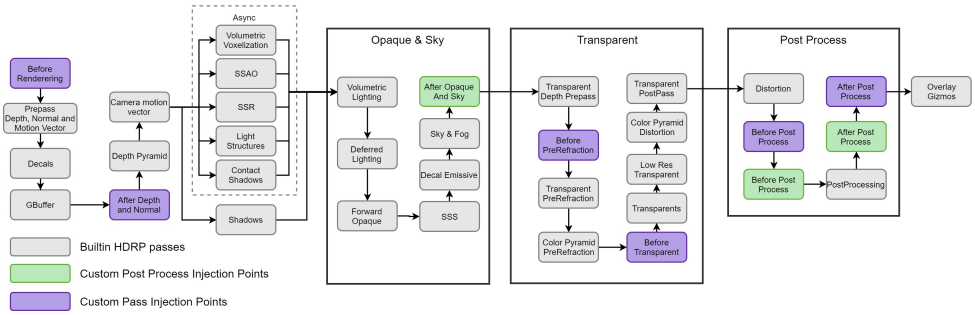


Figura 2.6: Pipeline de renderização de alta definição da Unity 3D.

A Figura 2.6 ilustra as etapas principais do pipeline de renderização de Alta Definição (HDRP) no Unity 3D, destacando o fluxo desde o pré-processamento até o pós-processamento. Dividido em fases como a renderização opaca, transparente e efeitos de pós-processamento, o HDRP permite a implementação de iluminação avançada, sombras

dinâmicas e efeitos visuais realistas. A figura também evidencia pontos de injeção personalizados para maior flexibilidade no desenvolvimento, oferecendo ferramentas para criar experiências visuais de alta qualidade e realismo.

Capítulo 3

Metodologia

Este capítulo discute a metodologia adotada no desenvolvimento de um simulador para aplicações de *visual servoing*, abordando os processos essenciais envolvidos em sua concepção. Inicialmente, será explorada a preparação do modelo 3D, detalhando os passos necessários para a criação e adaptação do modelo utilizado na simulação. Em seguida, discutiremos a integração desse modelo à plataforma Unity 3D, que serviu como base para a visualização e controle dos cenários simulados.

Na sequência, será apresentado o desenvolvimento do pacote para ROS 2, que permite a comunicação entre o simulador e os algoritmos de controle utilizados. Também será abordada a configuração dos cenários de simulação, destacando as condições e elementos projetados para testar diferentes aspectos do *visual servoing*. Por fim, discutiremos as métricas empregadas para a avaliação de desempenho do algoritmo desenvolvido.

3.1 Preparação do Modelo 3D

A criação do modelo 3D do robô Denso VP6242 foi uma etapa crucial para o desenvolvimento do simulador. Inicialmente, foi obtido um modelo CAD disponível na internet. No entanto, modelos CAD não são ideais para texturização devido à ausência de uma malha adequada. Para resolver este problema, o modelo CAD foi convertido para uma malha poligonal, processo que resultou em uma malha excessivamente densa e com detalhes desnecessários em diversas áreas.

3.1.1 Conversão e Retopologia

A conversão do modelo CAD para uma malha foi realizada utilizando ferramentas de modelagem 3D, contudo, a malha resultante apresentava alta densidade de polígonos, o que comprometeu a eficiência na texturização e renderização. Para otimizar a malha, foi realizado um processo de retopologia, onde toda a malha do robô foi refeita com base na malha gerada a partir do CAD, reduzindo a densidade de polígonos sem perder detalhes

essenciais.

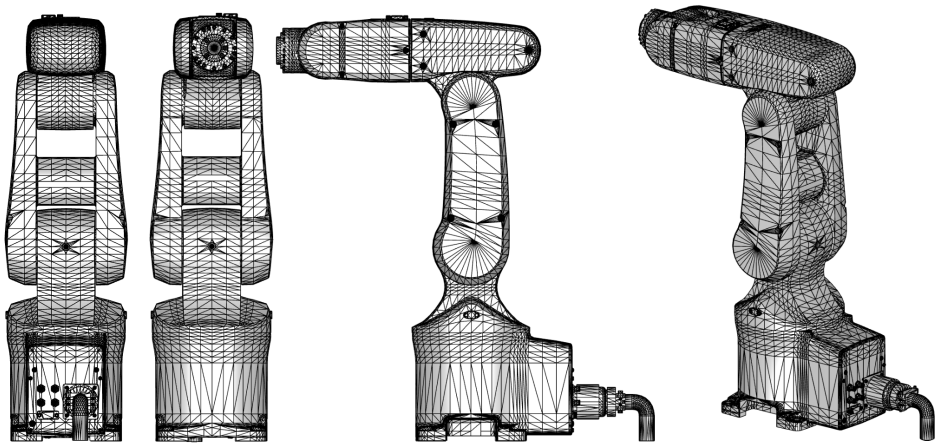


Figura 3.1: Retopologia aplicada ao modelo 3D do robô Denso VP6242.

3.1.2 Mapeamento UV e Texturização

Após a retopologia, foi necessário realizar o mapeamento UV, fundamental para a aplicação das texturas no modelo 3D. O mapeamento UV permite a projeção das texturas sobre a malha de forma correta, evitando distorções e sobreposições indesejadas. Com o mapeamento UV concluído, utilizou-se o Substance Painter para aplicar texturas detalhadas no modelo. Este software também foi empregado para realizar o *bake* dos mapas do modelo *high poly* para o *low poly*, preservando detalhes escondidos e otimizando a malha para utilização em tempo real.

Figura 3.2: Mapeamento UV aplicado ao modelo 3D do robô Denso VP6242.

3.1.3 Otimização da Malha

Para garantir que o modelo 3D fosse *game-ready*[1], realizou-se uma otimização adicional na malha, ajustando os níveis de detalhe e garantindo o desempenho na renderização dentro da Unity 3D. Este processo envolveu a redução de polígonos em áreas onde a alta densidade não era necessária, mantendo a qualidade visual do robô.

Figura 3.3: Modelo 3D texturizado do robô Denso VP6242.

3.2 Integração com Unity 3D

A Unity 3D foi escolhida como a plataforma de simulação devido à sua robustez e flexibilidade na criação de ambientes virtuais detalhados. Além disso, ela consegue exportar o jogo para diversas plataformas, facilitando o uso em variados dispositivos. A integração com a Unity 3D abrangeu não apenas a incorporação do modelo 3D do robô, mas também a configuração de sistemas de comunicação avançados utilizando ROS 2 e ROS-TCP-Connector, entre outros componentes essenciais. Esse processo envolveu várias etapas, desde a importação e otimização do modelo 3D até a implementação de uma comunicação bidirecional eficiente com o ROS 2, garantindo que todas as funcionalidades do robô fossem adequadamente representadas e controladas no ambiente virtual.

3.2.1 Importação do URDF

Utilizou-se o pacote URDF Importer[2] desenvolvido pela Unity Technologies para importar o URDF (*Unified Robot Description Format*) do robô Denso VP6242. O URDF original

[1]Pronto para ser utilizado em ambientes de jogos, com otimizações que asseguram desempenho eficiente e compatibilidade com motores de jogo como a Unity 3D.

[2]GitHub do URDF-Importer

foi adaptado substituindo os modelos 3D utilizados pelo modelo otimizado, permitindo a correta representação das juntas e componentes do robô na Unity 3D.

3.2.2 Configuração do ROS-TCP-Connector

Para estabelecer a comunicação entre a Unity 3D e o ROS 2, utilizou-se o pacote ROS-TCP-Connector[3], que permite a transferência de dados via protocolo TCP/IP. Este conector facilita a publicação e subscrição de tópicos ROS 2 diretamente na Unity 3D, garantindo uma comunicação eficiente e de baixa latência.

Publishers e Subscribers

A comunicação entre o Unity 3D e o ROS 2 foi configurada com os seguintes tópicos e formatos de mensagens:

- **Estado Atual das Juntas** (`joint_states_topic`):
  - **Tipo de Mensagem**: `sensor_msgs/JointState.msg`
  - **Descrição**: Publica o estado atual das juntas do robô, incluindo posições, velocidades e esforços.
  - **Direção**: Publisher (Unity 3D para ROS 2)
- **Imagem RGB** (`camera_image_topic`):
  - **Tipo de Mensagem**: `sensor_msgs/Image.msg`
  - **Descrição**: Publica imagens RGB capturadas pela câmera simulada.
  - **Direção**: Publisher (Unity 3D para ROS 2)
- **Imagem de Profundidade** (`camera_depth_topic`):
  - **Tipo de Mensagem**: `sensor_msgs/Image.msg`
  - **Descrição**: Publica imagens de profundidade capturadas pela câmera simulada.
  - **Direção**: Publisher (Unity 3D para ROS 2)
- **Comando de Movimento das Juntas** (`joint_command_topic`):
  - **Tipo de Mensagem**: `sensor_msgs/JointState.msg`
  - **Descrição**: Recebe comandos de movimentação das juntas do ROS 2, que atualizam a posição das juntas no ambiente Unity 3D.
  - **Direção**: Subscriber (ROS 2 para Unity 3D)

[3]GitHub do ROS-TCP-Connector

3.2.3 Controle das Juntas

O controle das juntas do robô na Unity 3D é realizado utilizando um *script* que gerencia a comunicação com o ROS 2, ajustando as posições e velocidades das juntas com base nos comandos recebidos. Além disso, o *script* é responsável por enviar o estado atual das juntas ao ROS 2, permitindo um fluxo bidirecional de informações. A lógica do controle das juntas segue as seguintes etapas principais:

- **Inicialização da Cadeia de Articulações**: Todas as juntas móveis do robô são identificadas e configuradas com parâmetros específicos, como atrito, amortecimento angular e limites de força, para garantir um controle estável.
- **Publicação do Estado das Juntas**: O *script* coleta informações como posição, velocidade e esforço de cada junta e publica essas informações em formato compatível com o ROS 2.
- **Recebimento de Comandos**: Os comandos recebidos do ROS 2, contendo posições desejadas para as juntas, são interpretados e aplicados às articulações do robô, convertendo os valores recebidos em unidades apropriadas para o ambiente Unity 3D.

3.3 Desenvolvimento do Pacote para ROS 2

O pacote desenvolvido para ROS 2 integra a Unity 3D e o ROS 2 para controle do robô, implementando o algoritmo de *image-based visual servoing* (IBVS). A seguir, descrevem-se os principais componentes e etapas.

3.3.1 Arquitetura do Pacote ROS 2

O pacote é composto pelos seguintes módulos:

- **Recepção de Dados**: Subscritores para capturar imagens RGB e de profundidade, além do estado das juntas do robô.
- **Processamento de Imagens**: Algoritmos para extração de características visuais (*features*).
- **Controle de Juntas**: Implementação do algoritmo IBVS que calcula comandos baseados no erro visual.
- **Publicação de Comandos**: Envio de comandos de controle para as juntas via tópicos ROS.

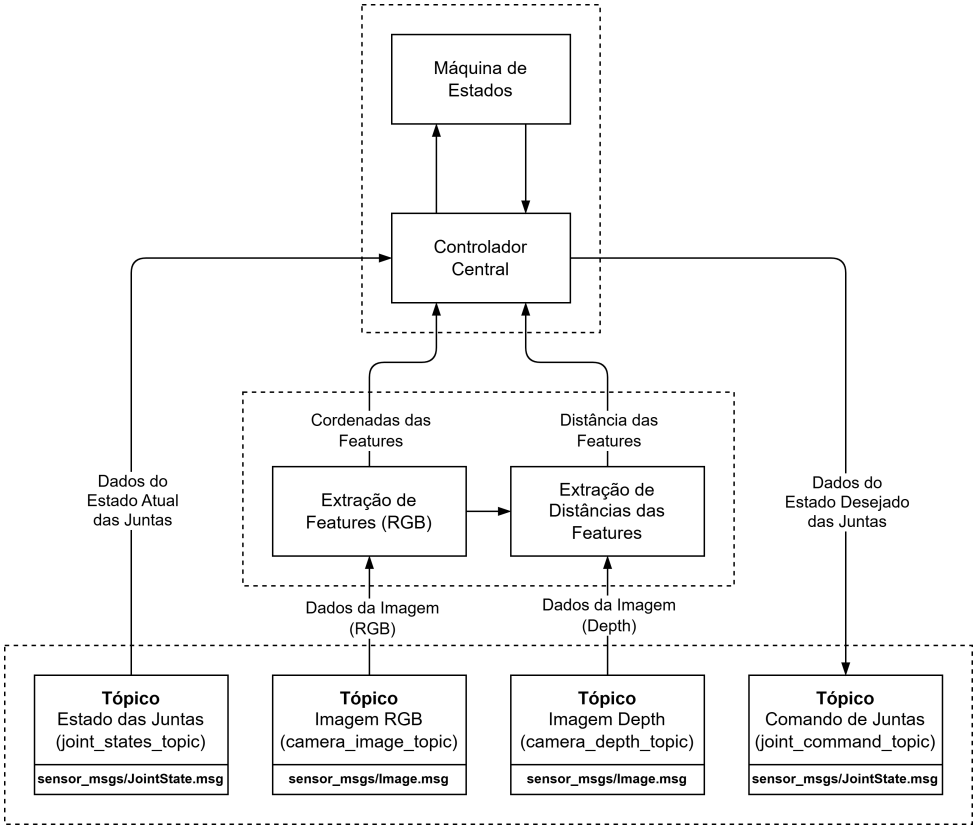


Figura 3.4: Fluxo de dados no pacote de controle desenvolvido para ROS 2.

3.3.2 Descrição dos Processos

Extração de Características Visuais

As características visuais são extraídas da imagem RGB utilizando processamento de imagens em OpenCV. As etapas incluem:

1. Conversão da imagem para escala de cinza.
2. Aplicação de limiares para destacar contornos.
3. Identificação das maiores áreas de interesse, como marcadores coloridos.
4. Cálculo das coordenadas centrais de cada região (momento geométrico).
5. Classificação das cores das características (vermelho, verde e azul).

Cálculo da Jacobiana da Imagem

A Jacobiana da imagem é calculada com base nas características extraídas:

$$J_p(u,v,Z) = \begin{bmatrix} -\hat{f}/Z & 0 & u/Z & uv/\hat{f} & -(\hat{f}+u^2)/\hat{f} & v \\ 0 & -\hat{f}/Z & v/Z & (\hat{f}+v^2)/\hat{f} & -uv/\hat{f} & -u \end{bmatrix}, \tag{3.1}$$

onde $\hat{f}$ é a distância focal, u, v são as coordenadas no plano da imagem, e Z é a profundidade do ponto.

Cálculo da Jacobiana do Robô

A Jacobiana do robô relaciona as velocidades articulares ($\dot{q}$) às velocidades no espaço operacional (v, ω). Para realizar este cálculo, utilizamos os parâmetros de Denavit-Hartenberg (DH), que fornecem uma maneira sistemática de descrever a geometria das juntas e dos elos do robô. A seguir, apresentamos os parâmetros DH utilizados para este robô:

Junta	θ_i (rad)	d_i (m)	a_i (m)	α_i (rad)
1	0	0.280	0	$-\frac{\pi}{2}$
2	$-\frac{\pi}{2}$	0	0.210	0
3	$-\frac{\pi}{2}$	0	-0.075	$-\frac{\pi}{2}$
4	0	0.210	0	$-\frac{\pi}{2}$
5	0	0	0	$\frac{\pi}{2}$
6	$\frac{\pi}{2}$	0.070	0	0

Tabela 3.1: Parâmetros de Denavit-Hartenberg (DH) do Robô

Com base nesses parâmetros DH, as transformações homogêneas T_i para cada junta são definidas como:

$$T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.2}$$

A matriz Jacobiana J é calculada propagando as transformações T_i para cada junta e considerando os vetores z_i e o_i obtidos a partir dessas transformações. A tabela 3.3.2 resume os parâmetros DH utilizados, facilitando a implementação e o cálculo da Jacobiana no espaço operacional do robô.

Cálculo do Erro Visual

O erro visual é calculado como a diferença entre as características desejadas (s^*) e as características atuais (s):

$$e = s^* - s. \tag{3.3}$$

Comando das Juntas

A pseudo-inversa da Jacobiana da imagem (J^+) é utilizada para calcular as velocidades desejadas da câmera:

$$v_{camera} = -\lambda J^+ e, \tag{3.4}$$

onde λ é o ganho proporcional. As velocidades articulares são determinadas por:

$$\dot{q} = J_{robot}^{-1} v_{camera}. \tag{3.5}$$

3.3.3 Loop Principal

Algorithm 1 Loop Principal do Controlador IBVS

```
1:  procedure ControlLoop
2:      Inicialização dos publishers e subscribers
3:      while nó ativo do
4:          features ← Extração de características visuais
5:          if features ≠ ∅ then
6:              e ← features_desired − features
7:              J_image ← Jacobiana da imagem
8:              J_robot ← Jacobiana do robô
9:              J^+ ← Pseudo-inversa de J_image
10:             v_camera ← −λJ^+ e
11:             q_vel ← J_robot^{-1} v_camera
12:             Publicar q_vel
13:         end if
14:     end while
15: end procedure
```

O loop principal do controlador IBVS, apresentado no Algoritmo 1, é responsável por integrar os módulos do sistema para realizar o controle do robô em tempo real. Ele realiza a aquisição contínua de dados, processa as informações recebidas, e aplica os cálculos necessários para gerar os comandos de movimento. O algoritmo implementa um fluxo lógico que inclui as seguintes etapas principais:

- Extração de características visuais (*features*) das imagens capturadas pela câmera RGB.
- Cálculo do erro visual com base na diferença entre as características atuais e as desejadas.

- Determinação da Jacobiana da imagem e do robô, necessárias para mapear as velocidades entre os espaços operacional e articular.
- Geração e publicação dos comandos de controle para alinhar o robô com o objetivo visual.

Durante cada iteração, o algoritmo verifica a presença de características visuais e, quando detectadas, realiza os cálculos necessários para determinar as velocidades articulares que minimizarão o erro visual. Este processo contínuo permite que o robô se adapte em tempo real às mudanças na cena observada.

3.4 Arquitetura do Sistema

Após detalhar a integração com a Unity 3D e o ROS 2, apresentamos a arquitetura geral do sistema proposto, que integra esses componentes com o pacote de controle de *visual servoing*.

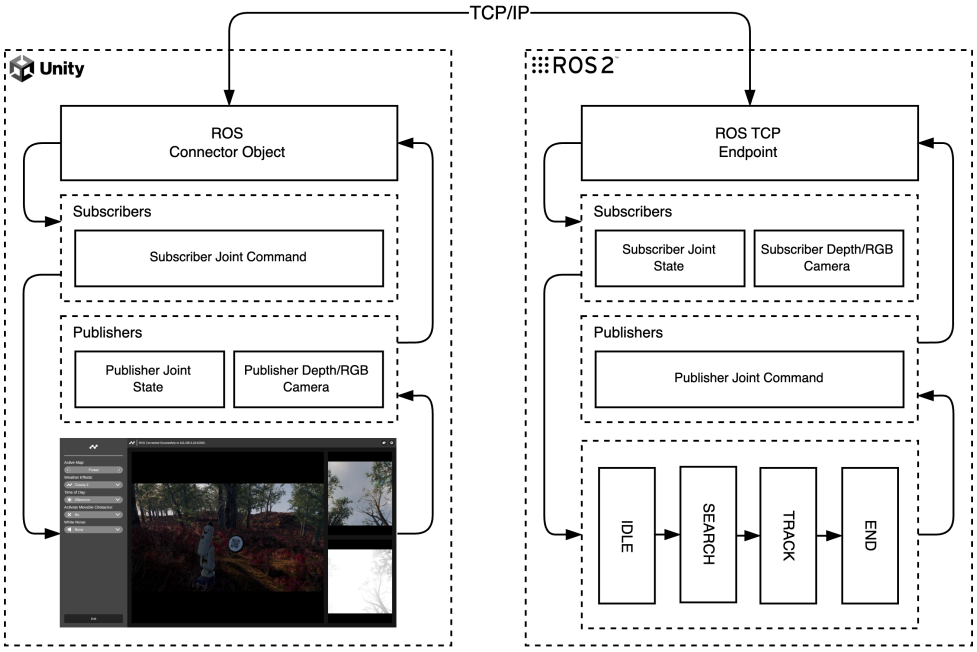


Figura 3.5: Arquitetura do sistema proposto.

A Figura 3.5 apresenta uma visão geral da arquitetura do sistema, ilustrando a comunicação entre os componentes e o fluxo de dados necessário para o controle em tempo real do robô. Observa-se como os módulos interagem para que o robô simulado na Unity 3D receba comandos e ajuste seu estado com base nas informações processadas pelo ROS 2, proporcionando uma estrutura integrada e eficiente para o desenvolvimento e validação de algoritmos de *visual servoing*.

3.5 Configuração dos Cenários de Simulação

Para avaliar a robustez e a eficiência do sistema de *visual servoing*, diferentes cenários de simulação foram configurados utilizando a plataforma Unity 3D. Cada cenário introduz variações específicas nas condições ambientais e nas tarefas a serem executadas pelo robô, permitindo uma análise abrangente do desempenho do sistema.

3.5.1 Descrição dos Cenários

Os cenários foram projetados para explorar diversas condições de operação, variando de ambientes ideais a situações mais desafiadoras. Essa abordagem possibilita a avaliação do desempenho do sistema em um espectro diversificado de configurações, simulando desafios que podem ocorrer em aplicações reais.

Cenário 1: Ambiente Ideal

Este cenário apresenta iluminação constante e ausência de obstáculos, criando um ambiente controlado e ideal para o teste preliminar do sistema. Ele permite verificar a funcionalidade básica do sistema sem interferências externas.

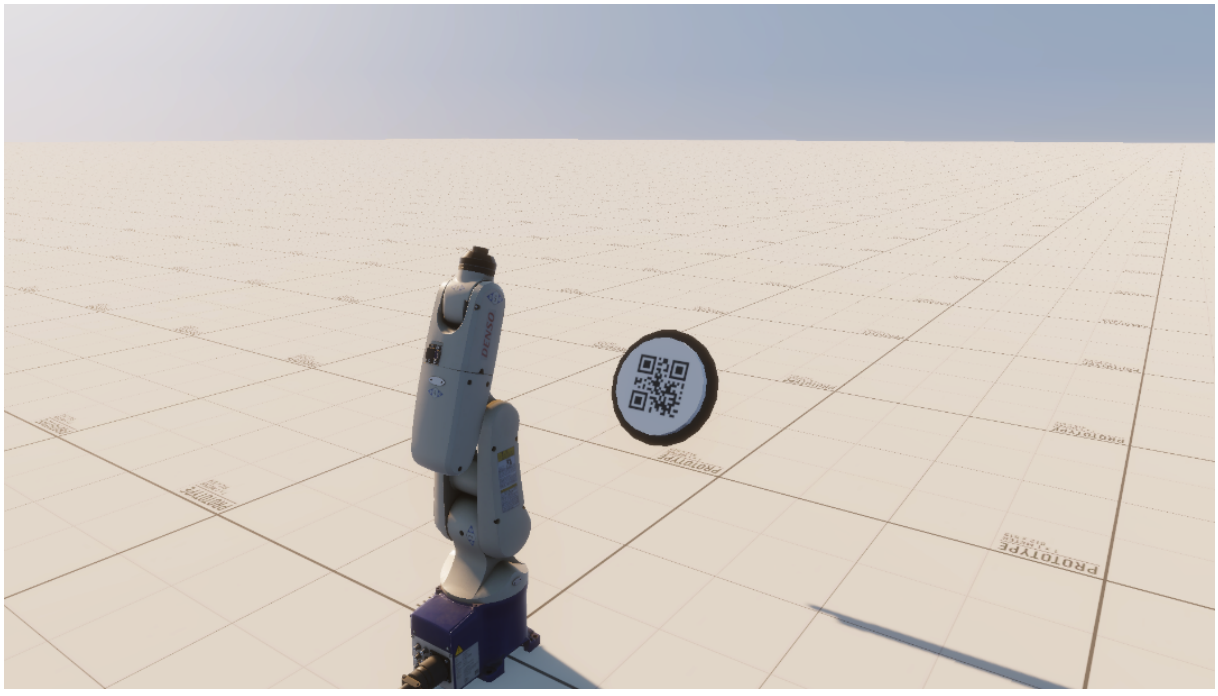


Figura 3.6: Cenário 1: Iluminação constante e ausência de obstáculos.

Cenário 2: Variação de Iluminação

Neste cenário, diferente do anterior, há sombras de árvores, plantas e grama ao fundo, com iluminação variável devido à passagem de nuvens, desafiando a robustez do sistema

em condições dinâmicas.

Figura 3.7: Cenário 2: Variação de iluminação.

Cenário 3: Adição de Ruído nas Imagens de Controle

Neste ambiente, ruídos são adicionados às imagens utilizadas para o controle do sistema, simulando condições de captura imperfeitas e interferências externas. Essa adição de ruído desafia a robustez do *visual servoing*, exigindo algoritmos mais sofisticados capazes de lidar com dados ruidosos e manter a precisão do controle mesmo em condições adversas.

Figura 3.8: Cenário 3: Adição de ruído nas imagens de controle.

3.5.2 Implementação das Variações Ambientais

As variações ambientais foram implementadas na Unity 3D, utilizando recursos avançados para simular condições realistas. Entre as funcionalidades, incluem-se ajustes dinâmicos na iluminação, adição de partículas para simular diferentes condições climáticas, como neve ou chuva, e *scripts* específicos para movimentação de objetos no ambiente. Estas variações são essenciais para avaliar a robustez e adaptabilidade do algoritmo de *visual servoing* em cenários desafiadores e próximos de situações reais, assegurando um controle preciso mesmo em ambientes adversos.

Figura 3.9: Exemplo de variação ambiental no cenário de simulação.

3.6 Metodologia

Nesta seção, são descritas as métricas utilizadas para avaliação do sistema de *visual servoing*, bem como os procedimentos experimentais adotados. As métricas escolhidas quantificam a precisão, eficiência e robustez do controle ao longo do tempo. A seguir, detalham-se os métodos e os critérios utilizados.

3.6.1 Erro Visual Residual Temporal ($e_v(t)$)

O erro visual residual temporal mede a diferença entre a posição atual das *features* visuais capturadas pela câmera do robô e a posição desejada dessas *features* na imagem de referência. A posição desejada é definida como o *target* centralizado no centro da imagem da câmera (neste caso, um QR Code). Essa métrica, expressa em pixels, é calculada ao longo

do tempo e avalia a capacidade do sistema de reduzir os desvios visuais e alinhar o alvo ao centro do campo de visão. Os valores coletados incluem a média e o desvio padrão, permitindo uma análise detalhada do desempenho do controle em diferentes condições experimentais.

3.6.2 Posições das Juntas do Robô ($q_i(t)$)

O comportamento das juntas do robô ao longo do tempo é analisado para verificar a estabilidade e o alinhamento com as posições desejadas. Essa métrica utiliza os valores das posições angulares das juntas ($q_i(t)$, onde i representa a junta), expressos em radianos. A análise dos valores médios e seus respectivos desvios padrão permite identificar instabilidades ou desvios em relação às trajetórias esperadas.

3.6.3 Proposta de Avaliação e Cenários

Para validar o sistema, foi realizada uma série de experimentos em diferentes cenários, variando condições iniciais e configurações do sistema. O robô foi inicializado em posições aleatórias, garantindo que o alvo estivesse visível a partir de cada posição inicial. A partir dessas condições, o sistema foi avaliado em 10 execuções por cenário, permitindo obter médias estatísticas dos resultados.

Cada experimento foi limitado a um tempo máximo de 60 segundos para evitar execuções indefinidamente prolongadas. Os valores das métricas foram coletados e comparados ao longo do tempo para identificar tendências, avaliar a eficiência do sistema e verificar sua robustez em diferentes condições experimentais.

A Tabela 3.2 apresenta um modelo ilustrativo dos valores esperados para as métricas coletadas durante os testes. Esses valores incluem o erro visual residual ($e_v(t)$) em pixels e as posições das juntas ($q_i(t)$) em radianos, avaliados em instantes específicos do tempo.

Cenário	**Tempo (s)**	$e_v(t)$ (px)	**Posições das Juntas ($q_i(t)$, rad)**					
			J1	J2	J3	J4	J5	J6
1	0.0	200.5	0.12	-0.08	0.05	0.00	0.03	0.02
	1.0	180.2	0.10	-0.06	0.04	0.01	0.05	0.04
	5.0	100.0	0.08	-0.02	0.02	0.03	0.07	0.06
2	0.0	220.0	0.15	-0.10	0.06	0.02	0.01	0.03
	1.0	190.4	0.12	-0.08	0.05	0.03	0.02	0.04
	5.0	110.8	0.10	-0.04	0.03	0.04	0.05	0.07
3	0.0	240.1	0.18	-0.12	0.08	0.04	0.02	0.05
	1.0	200.3	0.14	-0.10	0.06	0.05	0.03	0.06
	5.0	120.6	0.12	-0.06	0.04	0.06	0.04	0.08

Tabela 3.2: Modelo compacto dos valores das métricas coletadas.

Os cenários propostos consideraram fatores como posições iniciais do robô, a complexidade do ambiente simulado e o comportamento esperado do sistema de controle. Esses testes fornecem uma base para análises futuras e para melhorias no algoritmo de *visual servoing*.

Capítulo 4

Resultados

Este capítulo apresenta os resultados obtidos durante o desenvolvimento e a validação da aplicação de *image-based visual servoing* (IBVS). Inicialmente, descreve a aplicação final e suas funcionalidades. Posteriormente, discute os desafios enfrentados durante o desenvolvimento e as soluções implementadas. Por fim, apresenta a validação do algoritmo IBVS com métricas específicas, gráficos ilustrativos e análises das diferenças observadas.

4.1 Apresentação da Aplicação Final

A aplicação desenvolvida integra Unity 3D ao ROS 2 para simulação e controle de um robô com *visual servoing*. A Figura 4.1 mostra a interface principal com o ambiente simulado, o robô e o ponto de referência utilizado no *visual servoing*.

Figura 4.1: Interface da aplicação final de *visual servoing*.

O código-fonte da aplicação está disponível publicamente no repositório GitHub, permitindo que pesquisadores e desenvolvedores possam explorar e expandir as funcionalidades do simulador[1].

4.2 Validação do Algoritmo IBVS

A validação do algoritmo IBVS foi realizada por meio de experimentos em três cenários simulados com condições distintas: ambiente ideal, variação de iluminação e adição de ruído nas imagens de controle. As métricas avaliadas incluíram o erro visual residual temporal ($e_v(t)$) e as posições das juntas do robô ($q_i(t)$).

4.2.1 Cenário 1: Ambiente Ideal

No ambiente ideal, o erro visual apresentou duas características distintas dependendo da abordagem adotada para a geração dos gráficos. Na Figura 4.2, uma única plotagem demonstra uma convergência rápida do robô, evidenciando um gráfico monótono decrescente e estável. Este comportamento idealizado reflete as condições controladas onde a manipulabilidade do robô foi adequadamente gerenciada.

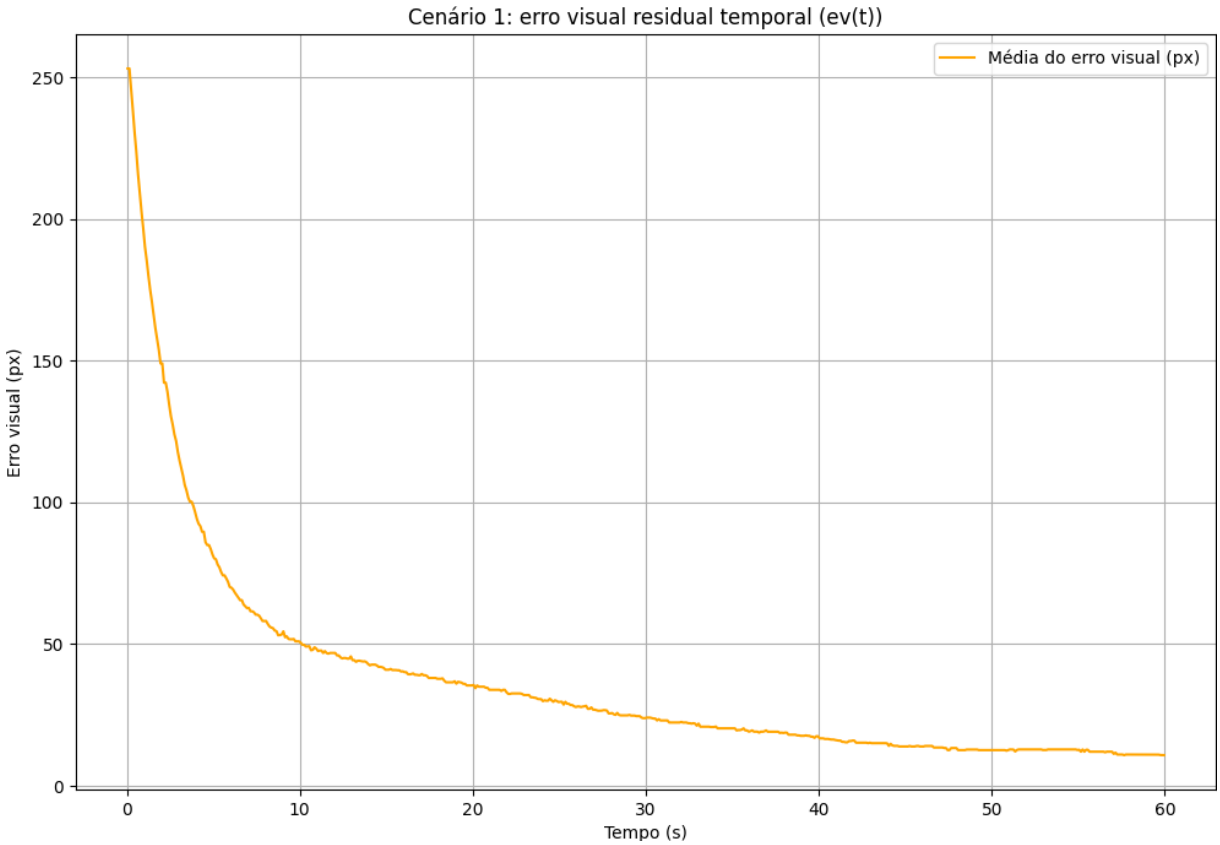


Figura 4.2: Cenário 1 ($e_v(t)$): Ambiente ideal (melhor resultado).

[1] Código fonte disponível no GitHub

Por outro lado, na Figura 4.3, apresenta-se uma condensação de vários testes, onde alguns experimentos se estenderam por até 60 segundos enquanto outros não. Este conjunto de dados resultou em gráficos que exibem mudanças abruptas ao longo do tempo. Tais variações ocorrem devido à falta de tratamento da manipulabilidade do robô, fazendo com que em diversos momentos o robô alcançasse posições onde não conseguia se mover mais, resultando na perda do tracking das *features*.

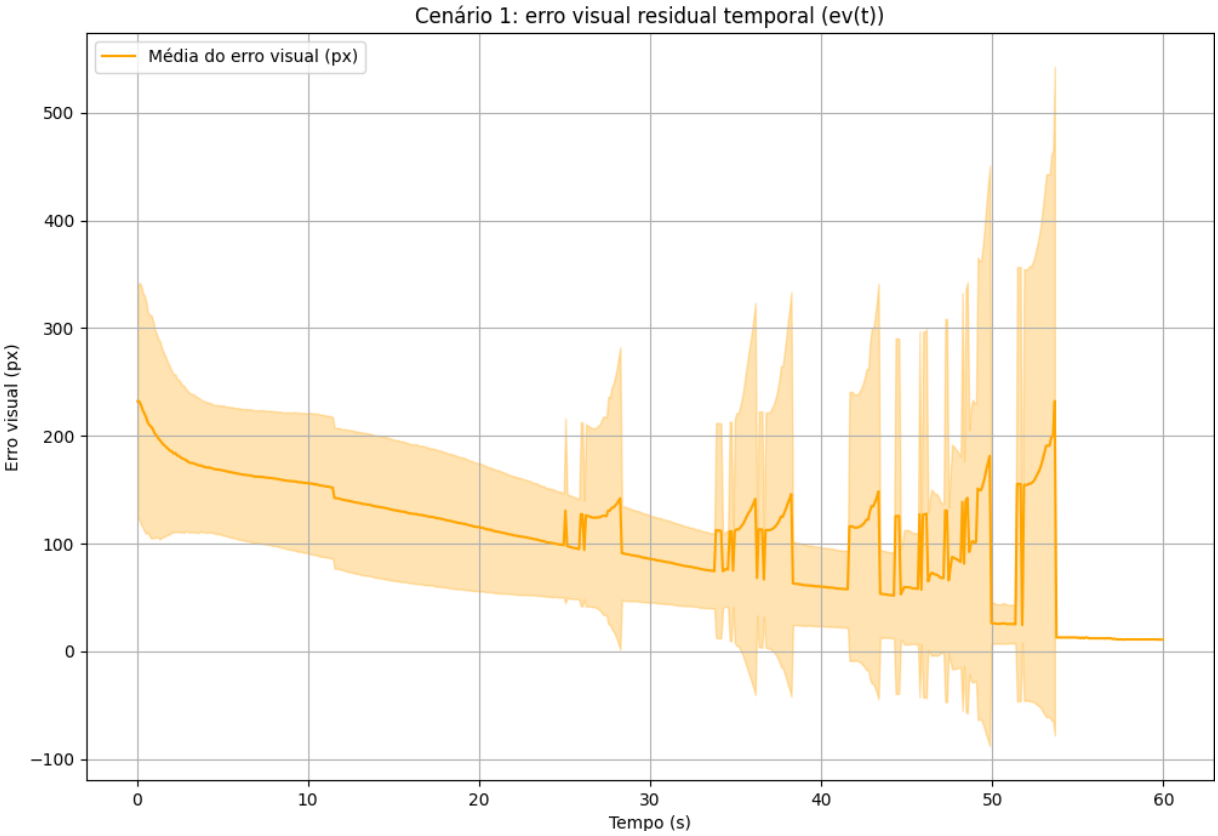


Figura 4.3: Cenário 1 ($e_v(t)$): Ambiente ideal (vários testes condensados).

As posições das juntas também refletem de forma clara as dinâmicas do sistema durante os experimentos realizados. Na Figura 4.4, é possível observar um comportamento geral estável, caracterizado por pequenas variações que indicam a eficiência do controle, mesmo diante das oscilações no erro visual. Esse cenário demonstra que as juntas se mantiveram consistentemente próximas aos valores desejados, confirmando um bom desempenho do sistema. Por outro lado, a Figura 4.5 apresenta um panorama distinto, onde a condensação de múltiplos testes evidencia variações expressivas e a ocorrência de picos de instabilidade, deixando evidente os momentos em que o robô perdeu o *tracking* e o erro aumentou de forma abrupta, afetando negativamente a precisão do movimento.

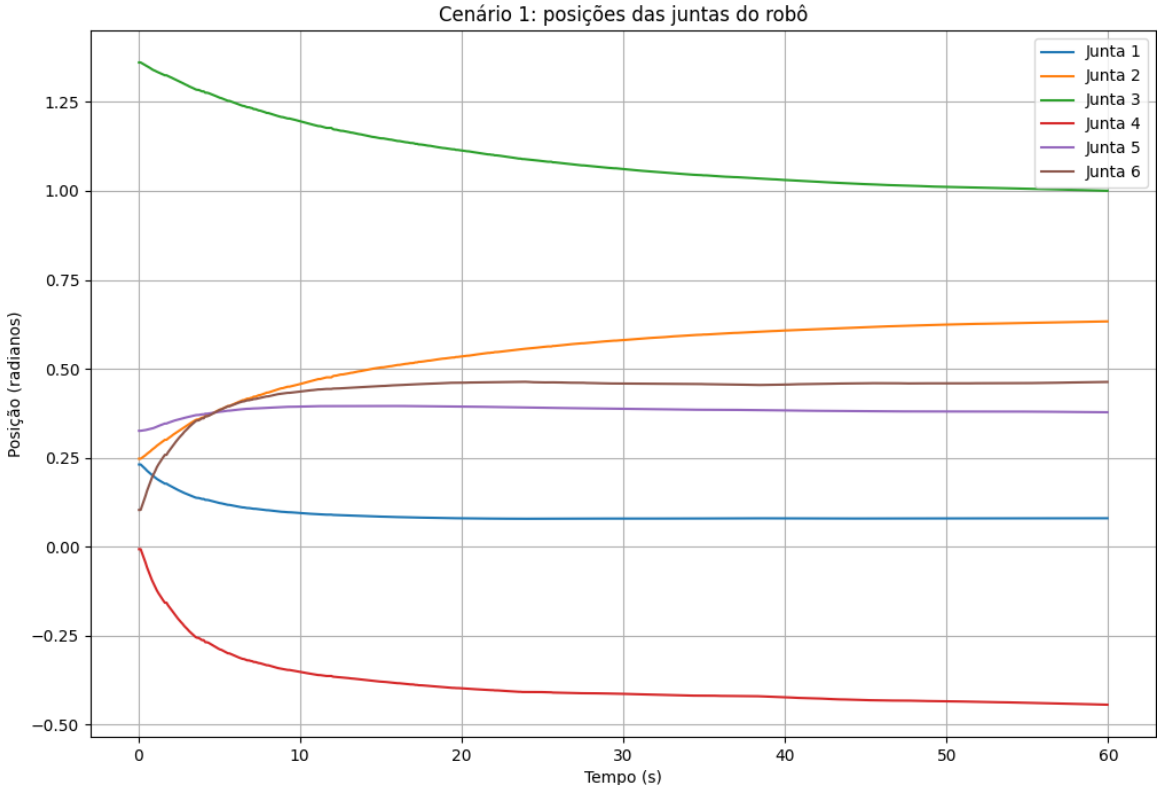


Figura 4.4: Cenário 1 ($q_i(t)$): Ambiente Ideal (melhor resultado.

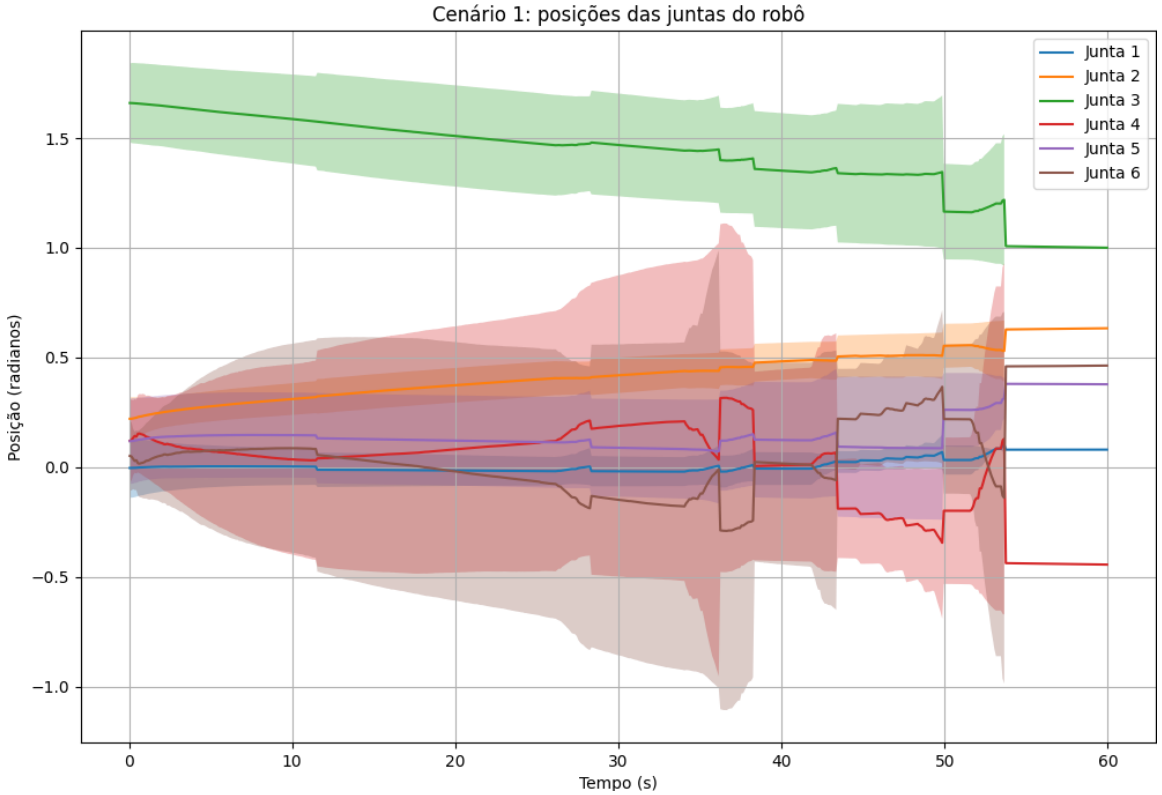


Figura 4.5: Cenário 1 ($q_i(t)$): Ambiente ideal (vários testes condensados).

Esses resultados indicam que, embora o algoritmo tenha alcançado seus objetivos

funcionais em condições ideais, a falta de tratamento adequado da manipulabilidade do robô resulta em perda de tracking e instabilidades, especialmente quando se considera a variabilidade entre diferentes testes.

4.2.2 Cenário 2: Variação de Iluminação

No Cenário 2, com variações de luz e sombras, o sistema enfrentou instabilidade e flutuações no erro visual, como mostrado na Figura 4.6. A condensação de vários testes, incluindo aqueles com durações variáveis, levou a gráficos com mudanças abruptas ao longo do tempo. Essas variações são consequência da manipulação não tratada da manipulabilidade do robô, que frequentemente atingia posições onde não conseguia se mover, resultando na perda do tracking das *features*.

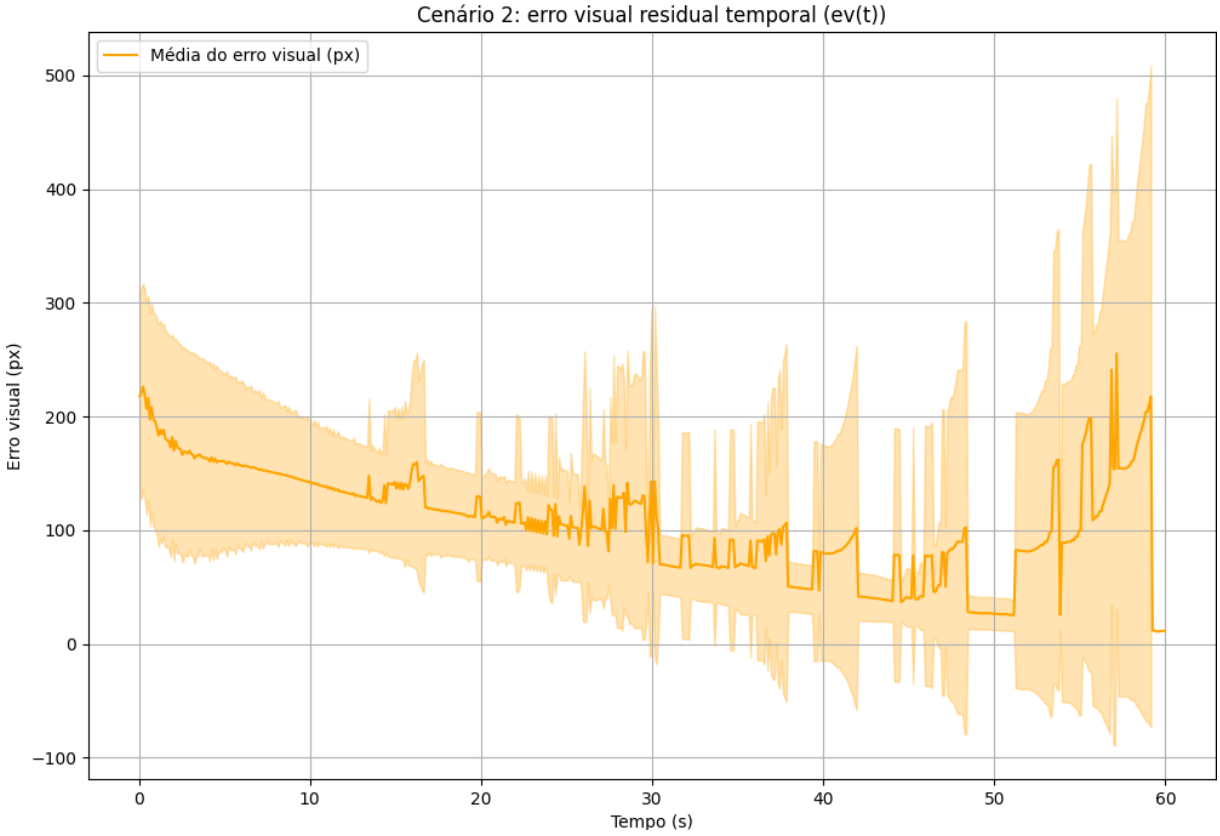


Figura 4.6: Cenário 2 ($e_v(t)$): Variação de iluminação (vários testes condensados).

Além disso, as condições de variação luminosa, como sombras, agravaram a perda de tracking, pois as *features* tornaram-se menos distinguíveis. A Figura 4.7 ilustra como as posições das juntas foram afetadas, exibindo oscilações significativas que refletem a dificuldade do algoritmo em manter a estabilidade do controle sob mudanças ambientais constantes. A perda recorrente de tracking obrigou o robô a se reposicionar repetidamente, causando picos de erro e comprometendo o desempenho geral do sistema.

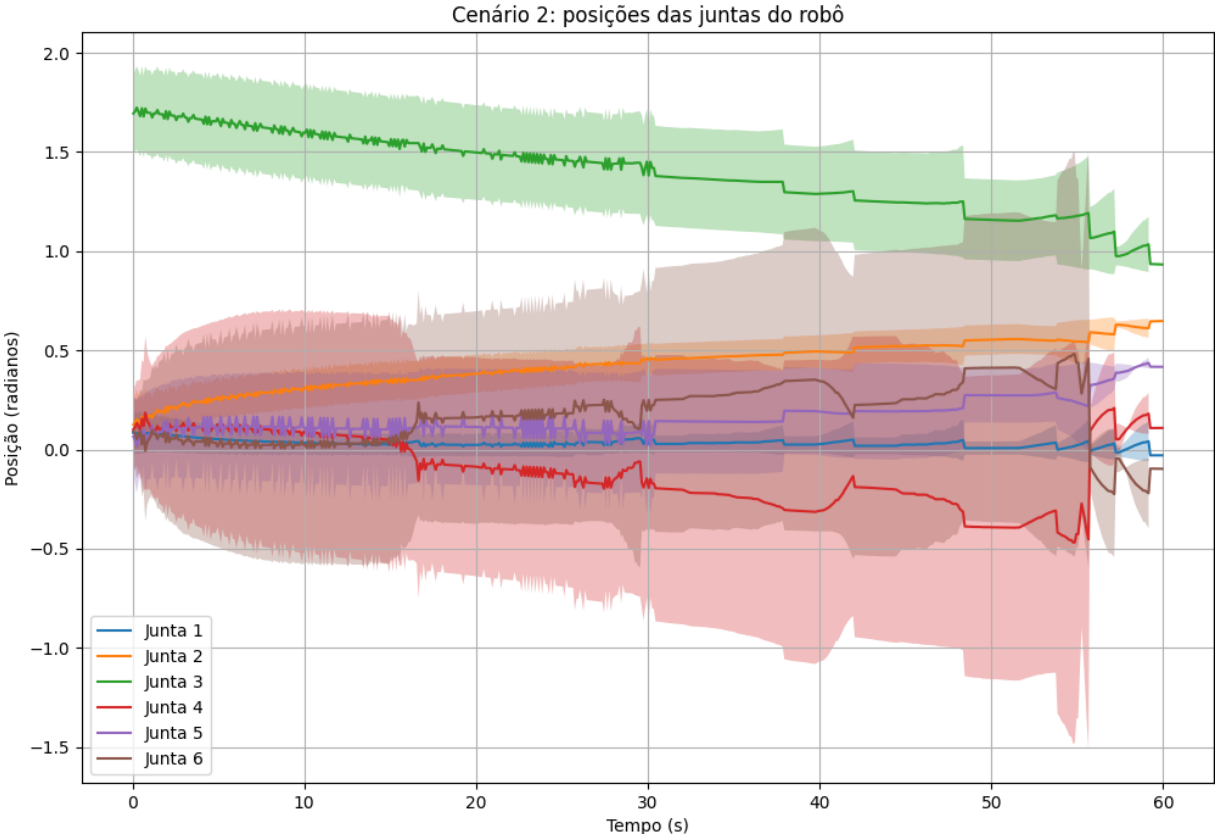


Figura 4.7: Cenário 2 ($q_i(t)$): Variação de iluminação (vários testes condensados).

Embora os valores absolutos do desvio padrão das posições articulares sejam pequenos, em radianos, o aumento em relação ao Cenário 1 demonstra uma alta variação relativa, diretamente influenciada pela escala angular adotada. Essa maior variabilidade aponta para oscilações que podem comprometer tanto a precisão quanto a estabilidade do sistema em aplicações práticas. A interação entre as condições ambientais adversas e a falta de robustez na manipulabilidade do robô resultou em um desempenho inferior, embora o controle básico das juntas tenha sido mantido.

Esse comportamento ressalta a importância de projetar algoritmos de controle mais robustos a variações ambientais. Estratégias como a utilização de técnicas de aprendizado de máquina para identificação adaptativa de *features* visuais ou o emprego de sensores adicionais para complementar a percepção podem mitigar esses efeitos adversos. Além disso, ajustes no pré-processamento das imagens, como normalização de contraste ou filtros específicos para redução de ruídos, poderiam melhorar a detecção das *features*, diminuindo a frequência de perdas de tracking. Esses aprimoramentos seriam cruciais para aumentar a confiabilidade do sistema em cenários mais desafiadores, especialmente quando aplicados a ambientes dinâmicos e não controlados.

4.2.3 Cenário 3: Adição de Ruído nas Imagens de Controle

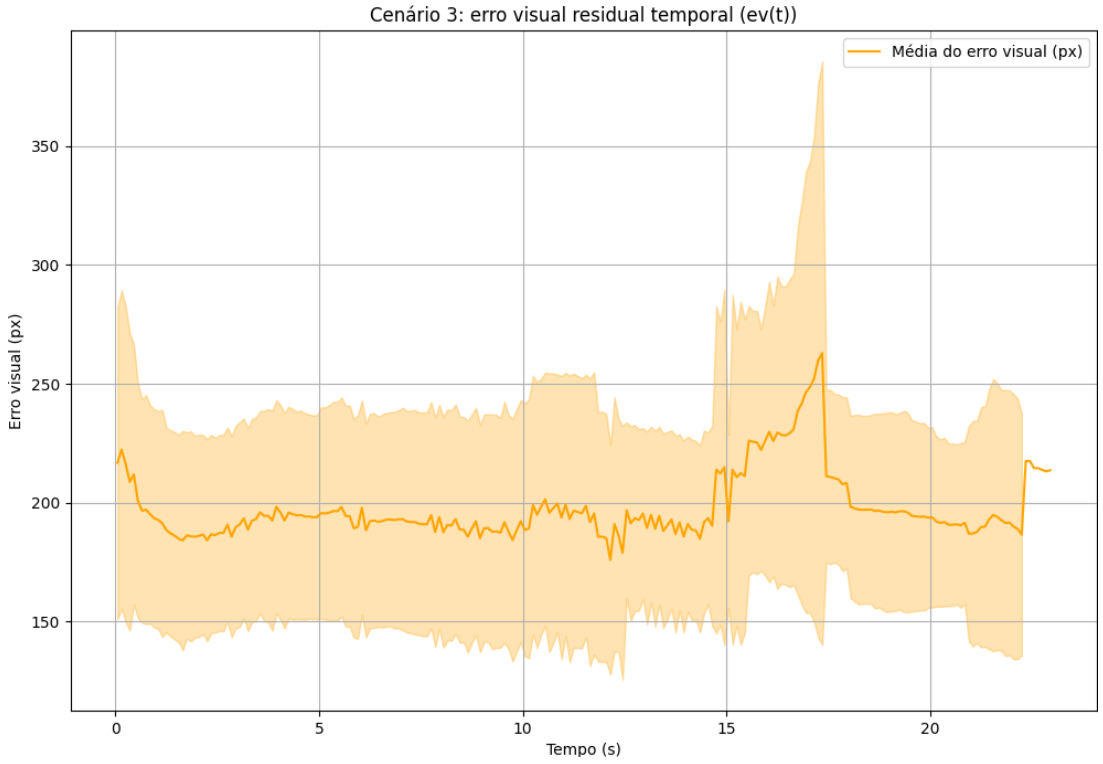


Figura 4.8: Cenário 3 ($e_v(t)$): Adição de ruído (vários testes condensados).

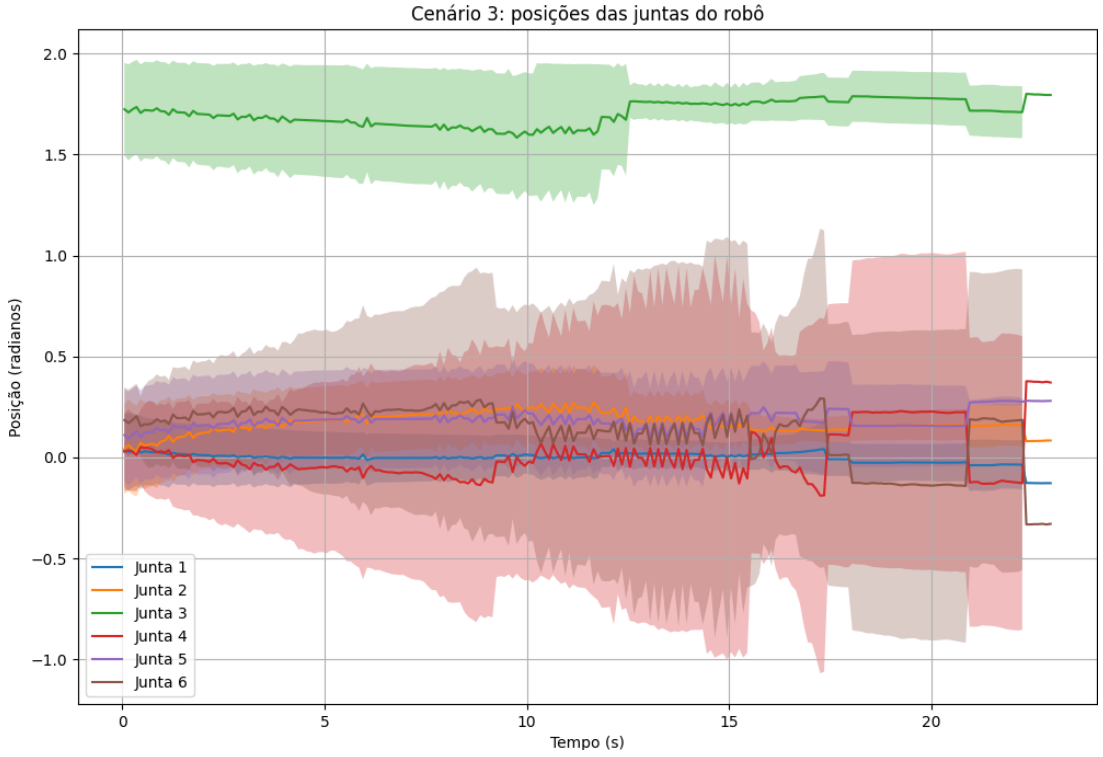


Figura 4.9: Cenário 3 ($q_i(t)$): Adição de ruído (vários testes condensados).

No Cenário 3, a adição de ruído nas imagens de controle exacerbou os problemas observados nos cenários anteriores. A Figura 4.8 demonstra que o erro visual residual não apenas flutuou significativamente, mas também não conseguiu atingir a convergência estável observada na Figura 4.2. A presença de ruído dificultou a correta captura das *features*, levando à perda de tracking frequente e, consequentemente, à necessidade de reinicializações do sistema.

As posições das juntas, conforme ilustrado na Figura 4.9, exibem grandes oscilações e instabilidades contínuas. O ruído impediu a obtenção precisa das *features*, o que, aliado à manipulação não tratada, causou movimentos descontrolados do robô e aumento abrupto do erro. No Cenário 3, o problema do ruído foi o fator predominante que impossibilitou a operação estável do algoritmo IBVS, refletindo-se diretamente na perda de tracking e na ineficiência do controle das juntas.

Esses resultados evidenciam que, em condições adversas como a adição de ruído, o algoritmo IBVS requer mecanismos adicionais para manter a estabilidade e a precisão do controle, especialmente no que diz respeito à manipulabilidade e ao tracking das *features*.

4.3 Melhorias Propostas

Com base nos resultados, propõe-se a implementação de técnicas avançadas de filtragem, como o RMCKF (*Regularized Maximum Correntropy Criterion Kalman Filter*), para mitigar o impacto de ruídos nas estimativas das características visuais. Sugere-se também a adaptação dinâmica dos ganhos de controle, ajustando os parâmetros do IBVS conforme as condições ambientais, conforme descrito por Leite (2023). Adicionalmente, a utilização de algoritmos de visão computacional mais robustos, como redes neurais treinadas para detecção e rastreamento de características visuais em cenários adversos, pode trazer melhorias significativas. Para complementar os dados visuais e aumentar a precisão do sistema, a fusão sensorial, integrando informações de sensores como LIDAR e IMU, é uma alternativa promissora.

Além disso, é fundamental tratar a manipulabilidade do robô de forma mais robusta, implementando estratégias que evitem que o robô atinja posições onde não possa se mover adequadamente, prevenindo assim a perda de tracking. A implementação de algoritmos de detecção e recuperação de falhas de tracking pode ajudar o sistema a reconhecer e corrigir automaticamente esses estados, garantindo maior estabilidade e continuidade no controle. Essas medidas visam aprimorar a robustez e a eficiência do sistema IBVS em ambientes reais, garantindo maior estabilidade e confiabilidade frente a condições adversas.

Capítulo 5

Conclusão

Este trabalho apresentou o desenvolvimento de um simulador de controle visual baseado em imagem, utilizando a plataforma Unity 3D integrada ao *framework* ROS 2 por meio do pacote ROS-TCP-Connector. A solução demonstrou a viabilidade de realizar simulações em ambientes virtuais, integrando controle robótico e captura de imagens, oferecendo uma plataforma para validação de algoritmos de controle visual em situação ideal e situações desafiadoras, como ruído e intempéries simulados.

O sistema desenvolvido mostrou-se capaz de reproduzir cenários complexos, como variações de iluminação, obstáculos e ruídos nas imagens, fornecendo uma base para análise e validação. A arquitetura modular garantiu comunicação entre o simulador e o ROS 2, além de flexibilidade para futuras expansões. O Unity 3D destacou-se pela criação de ambientes realistas para experimentos e pesquisas.

Apesar dos resultados positivos, a abordagem adotada apresenta algumas limitações. A execução do simulador no Unity 3D requer recursos computacionais elevados, o que pode limitar seu uso em máquinas com especificações menores. Além disso, a comunicação entre o Unity 3D e o ROS 2 depende do ROS-TCP-Connector e do ROS-TCP-Endpoint, introduzindo complexidade adicional e possíveis latências que afetam a performance em tempo real. Ademais, existe a lacuna entre a simulação e o ambiente real (*sim-to-real gap*), exigindo ajustes ao implementar os algoritmos em robôs físicos. Por fim, a escalabilidade pode ser um desafio ao adicionar robôs mais complexos ou ambientes detalhados, necessitando otimizações contínuas para manter a eficiência da plataforma.

Como extensão, propõe-se validar o sistema em robôs físicos, aproveitando o simulador como gêmeo digital para facilitar a transição entre os ambientes virtual e real. Além disso, a integração de métodos avançados, como aprendizado de máquina e abordagens híbridas de controle visual, pode aumentar a robustez e eficiência em cenários dinâmicos, considerando as métricas trazidas neste trabalho.

Conclui-se que o simulador desenvolvido contribui para a pesquisa em servovisão robótica, oferecendo uma plataforma acessível para o desenvolvimento, teste e validação de algoritmos, atendendo tanto às necessidades acadêmicas quanto às demandas industriais.

Bibliografia

Akenine-Möller, T., Haines, E., and Hoffman, N. (2018). *Real-Time Rendering, Fourth Edition*. CRC Press, 4 edition.

Balogh, R. and Obdržálek, D. (2019). Using finite state machines in introductory robotics. In Lepuschitz, W., Merdan, M., Koppensteiner, G., Balogh, R., and Obdržálek, D., editors, *Robotics in Education*, pages 85–91, Cham. Springer International Publishing.

Corke, P. (2023). *Robotics, Vision and Control: Fundamental Algorithms in Python*. Springer International Publishing.

Craig, J. J. (2016). *Introduction to robotics*. Pearson, Upper Saddle River, NJ, 4 edition.

de Melo, M. S. P., de Melo, M. S. P., da Silva Neto, J. G., da Silva Neto, J. G., da Silva, P. J. L., da Silva, P. J. L., Teixeira, J. M. X. N., Teixeira, J. M., Teichrieb, V., and Teichrieb, V. (2019). Analysis and comparison of robotics 3d simulators. *Symposium on Virtual and Augmented Reality*.

Flavell, L. (2010). *UV Mapping*, pages 97–122. Apress, Berkeley, CA.

Hearn, D., Baker, M., and Carithers, W. (2011). *Computer Graphics with OpenGL*. Prentice Hall, 4 edition.

Hu, J., Lu, Q., Fan, H., Xiao, Y., Zhou, Y., and Zhang, S. (2024). Unity 3d-based six-degree-of-freedom robotic arm virtual simulation teaching platform. *2024 3rd International Conference on Robotics, Artificial Intelligence and Intelligent Control (RAIIC)*.

Hughes, J. (2014). *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 3 edition.

Kim, D., Kim, D.-J., Lovelett, R., Lovelett, R., Behal, A., and Behal, A. (2009). Eye-in-hand stereo visual servoing of an assistive robot arm in unstructured environments. *IEEE International Conference on Robotics and Automation*.

Leite, G. R., Araújo, . B. Q. d., and Martins, A. d. M. (2023). Regularized maximum correntropy criterion kalman filter for uncalibrated visual servoing in the presence of non-gaussian feature tracking noise. *Sensors*, 23(20).

Li, S., Li, S., Xie, W., Xie, W.-F., Gao, Y., Gao, Y., and Gao, Y. (2017). Enhanced ibvs controller for a 6dof manipulator using hybrid pd-smc method. *International Journal of Control, Automation and Systems*.

Lynch, K. M. and Park, F. C. (2017). *Modern robotics*. Cambridge University Press, Cambridge, England.

Marshall, M. and Lipkin, H. (2014). Kalman filter visual servoing control law. In *2014 IEEE International Conference on Mechatronics and Automation*, page 527–5324. IEEE.

Mattingly, W. A., Mattingly, W. A., Chang, D.-J., Chang, D.-J., Paris, R., Paris, R., Paris, R., Smith, N., Smith, N., Blevins, J. C., Blevins, J., Ouyang, M., and Ouyang, M. (2012). Robot design using unity for computer games and robotic simulations. *International Conference on Computer Games*.

Piegl, L. and Tiller, W. (1996). *The NURBS Book*. Monographs in Visual Communication. Springer, Berlin, Germany, 2 edition.

Platt, J. and Ricks, K. G. (2022). Comparative analysis of ros-unity3d and ros-gazebo for mobile ground robot simulation. *Journal of Intelligent & Robotic Systems*.

Rost, R. J., Liceakane, B., Ginsburg, D., Kessenich, J., Lichtenbelt, B., Malan, H., and Weiblen, M. (2009). *OpenGL Shading Language (3rd Edition)*. Addison-Wesley Professional, 3 edition.

Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. (2009). *Robotics*. Springer London.

Spong, M. W., Hutchinson, S., and Vidyasagar, M. (2020). *Robot modeling and control*. John Wiley & Sons, Nashville, TN, 2 edition.

Yu, Z. (2022). Ray tracing in computer graphics. *Journal of Computer Graphics Techniques*, 24:99–106.