

**UNIVERSIDADE FEDERAL DE UBERLÂNDIA  
FACULDADE DE ENGENHARIA ELÉTRICA  
GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO**

**MATHEUS MENEZES PEDROSA**

**IMPLEMENTAÇÃO DO ALGORITMO HYBRID A\* PARA O  
PLANEJAMENTO DE TRAJETÓRIAS DE VEÍCULOS  
AUTÔNOMOS EM AMBIENTE VIRTUAL**

**UBERLÂNDIA, MG  
2024**

Matheus Menezes Pedrosa

# **Implementação do Algoritmo Hybrid A\* para o Planejamento de Trajetórias de Veículos Autônomos em Ambiente Virtual**

Trabalho de Conclusão de Curso da Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Campus Santa Mônica, como requisito para a obtenção do título de Graduação em Engenharia de Controle e Automação.

Universidade Federal de Uberlândia - UFU  
Faculdade de Engenharia Elétrica - FEELT

Orientador Prof. Me. Éder Alves de Moura

Uberlândia, MG

2024

# Agradecimentos

Expresso minha mais sincera gratidão à minha família – meus pais, Renato e Adrielle, meus avós, Elenice, Jaci e Neidimar, e meus tios, Rodrigo, Robson e Regis – pelo suporte que foi essencial ao longo de toda a minha vida.

Agradeço também aos amigos que fiz ao longo desta jornada, tanto na faculdade quanto fora dela. Cada um de vocês contribuiu para tornar este percurso mais leve e repleto de momentos inesquecíveis.

Minha gratidão se estende aos professores que contribuíram para minha formação, especialmente ao professor Éder Alves de Moura, por sua orientação dedicada e decisiva na reta final deste trabalho.

Por fim, agradeço a mim mesmo, pela resiliência, pelo esforço constante e pela capacidade de superar desafios. Este trabalho é reflexo direto de minhas escolhas e da determinação que me trouxe até aqui.

# Resumo

Este trabalho propõe a implementação e análise do algoritmo Hybrid A\* para o planejamento de trajetórias em veículos autônomos, utilizando a plataforma Unity 3D como ambiente virtual de simulação. A pesquisa explora as capacidades do Unity para criar simulações realistas, permitindo avaliar a eficácia e precisão do algoritmo em condições variadas e complexas. O algoritmo Hybrid A\*, uma extensão do tradicional A\*, é desenvolvido especificamente para lidar com restrições não-holonômicas, essenciais em cenários de navegação de veículos autônomos. Embora seu desempenho dependa amplamente de uma heurística eficiente, a implementação mostra que o Hybrid A\* é eficaz em gerar rotas viáveis e seguras, respeitando as limitações cinemáticas do veículo. A criação do ambiente virtual no Unity, onde são simulados obstáculos e condições realistas de navegação, permite o teste e a avaliação de trajetórias seguras e viáveis. O estudo contribui para o avanço na área de veículos autônomos, destacando o potencial do Unity como ferramenta para o desenvolvimento e teste de algoritmos de planejamento de trajetórias.

**Palavras-chaves:** Hybrid A\*; Planejamento de Trajetórias; Veículos Autônomos; Unity 3D; Ambiente Virtual.

# Abstract

This study proposes the implementation and analysis of the Hybrid A\* algorithm for path planning in autonomous vehicles, using the Unity 3D platform as a simulation environment. The research explores Unity's capabilities to create realistic simulations, enabling the assessment of the algorithm's effectiveness and accuracy under varied and complex conditions. The Hybrid A\* algorithm, an extension of the traditional A\*, is specifically designed to address non-holonomic constraints, which are essential in autonomous vehicle navigation scenarios. Although its performance largely depends on an efficient heuristic, the implementation demonstrates that Hybrid A\* is effective in generating feasible and safe routes, respecting the vehicle's kinematic limitations. The creation of a virtual environment in Unity, where obstacles and realistic navigation conditions are simulated, allows for the testing and evaluation of safe and viable trajectories. This study contributes to advancements in the field of autonomous vehicles, highlighting Unity's potential as a tool for developing and testing path planning algorithms.

**Key-words:** Hybrid A\*; Path Planning; Autonomous Vehicle; Unity 3D; Virtual Environment.

# Lista de Figuras

|                                                                                                                                                                 |    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figura 1 – Modelo cinemático de um veículo do tipo carro, representado pelo modelo de bicicleta. . . . .                                                        | 16 |
| Figura 2 – Exemplo de <i>min-heap</i> para organização de prioridade. . . . .                                                                                   | 19 |
| Figura 3 – Exemplo de um Diagrama de Voronoi gerado a partir de um conjunto de pontos. . . . .                                                                  | 20 |
| Figura 4 – Exemplo de campo de fluxo gerado pelo algoritmo Brushfire para repulsão. . . . .                                                                     | 23 |
| Figura 5 – Exemplo de campo de fluxo gerado pelo algoritmo Brushfire para atração. . . . .                                                                      | 23 |
| Figura 6 – Exemplo de trajetórias baseadas nas curvas de Reeds-Shepp e Dubins. . . . .                                                                          | 28 |
| Figura 7 – Interface do Unity com painel de hierarquia, cena, inspetor e projeto. . . . .                                                                       | 29 |
| Figura 8 – Comparação gráfica entre algoritmos de busca: A* tradicional, A* com grade refinada, e Hybrid A*. . . . .                                            | 37 |
| Figura 9 – Exemplo de Expansão Analítica usando Reeds-Shepp. . . . .                                                                                            | 39 |
| Figura 10 – Exemplo do Campo de Voronoi aplicado em um ambiente com obstáculos. . . . .                                                                         | 40 |
| Figura 11 – Diagrama da implementação do algoritmo Hybrid A*. . . . .                                                                                           | 42 |
| Figura 12 – Dimensões do Tesla Model S. . . . .                                                                                                                 | 47 |
| Figura 13 – <i>Scripts</i> utilizados para a implementação do Hybrid A*. . . . .                                                                                | 48 |
| Figura 14 – Formação da grade com células de 0,5 m, mostrando dois blocos como obstáculos e as células bloqueadas em preto. . . . .                             | 51 |
| Figura 15 – Campo de Fluxo de atração com célula objetivo no centro em roxo, variando de verde (baixo custo) a vermelho (alto custo). . . . .                   | 53 |
| Figura 16 – Campo de Fluxo de atração com obstáculos, destacando áreas intransitáveis em azul. . . . .                                                          | 54 |
| Figura 17 – Mapa com aplicação do A*, célula inicial próxima ao objetivo e ao contorno em "C". . . . .                                                          | 55 |
| Figura 18 – Mapa com aplicação do A*, célula inicial distante do objetivo. . . . .                                                                              | 55 |
| Figura 19 – Regiões de influência dos obstáculos no Campo de Voronoi. . . . .                                                                                   | 57 |
| Figura 20 – Campo de Voronoi . . . . .                                                                                                                          | 58 |
| Figura 21 – Cenário de estacionamento com Expansão Analítica, mostrando os nós expandidos ao longo do caminho. . . . .                                          | 62 |
| Figura 22 – Cenário de estacionamento com Expansão Analítica, exibindo apenas a trajetória final encontrada. . . . .                                            | 62 |
| Figura 23 – Cenário de espaço confinado sem Expansão Analítica, mostrando os nós expandidos, evidenciando a grande quantidade de expansões necessárias. . . . . | 63 |
| Figura 24 – Cenário de espaço confinado sem Expansão Analítica, exibindo apenas a trajetória final encontrada. . . . .                                          | 63 |

|                                                                                                                                                                               |    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figura 25 – Cenário de espaço confinado com Expansão Analítica, mostrando os nós expandidos, com uma redução significativa em relação ao caso sem Expansão Analítica. . . . . | 64 |
| Figura 26 – Cenário de espaço confinado com Expansão Analítica, exibindo apenas a trajetória final encontrada. . . . .                                                        | 64 |
| Figura 27 – Cenário complexo com múltiplos obstáculos e um ponto de estacionamento apertado, mostrando os nós expandidos ao longo da trajetória. .                            | 65 |
| Figura 28 – Cenário complexo com múltiplos obstáculos e um ponto de estacionamento apertado, exibindo apenas a trajetória final encontrada. . . . .                           | 65 |

# Lista de Algoritmos

|   |                                                          |    |
|---|----------------------------------------------------------|----|
| 1 | Pseudocódigo para o Brushfire. . . . .                   | 25 |
| 2 | Pseudocódigo para o cálculo do Campo de Voronoi. . . . . | 26 |
| 3 | Pseudocódigo para o A*. . . . .                          | 35 |
| 4 | Pseudocódigo para o Hybrid A*. . . . .                   | 45 |

# Lista de tabelas

|                                                                                                             |    |
|-------------------------------------------------------------------------------------------------------------|----|
| Tabela 1 – Dimensões do Tesla Model S . . . . .                                                             | 47 |
| Tabela 2 – Organização dos <i>scripts</i> de acordo com as etapas de implementação do<br>Hybrid A*. . . . . | 49 |

# Sumário

|            |                                                                |           |
|------------|----------------------------------------------------------------|-----------|
| <b>1</b>   | <b>INTRODUÇÃO</b>                                              | <b>11</b> |
| <b>1.1</b> | <b>Contextualização</b>                                        | <b>11</b> |
| <b>1.2</b> | <b>Objetivos</b>                                               | <b>12</b> |
| <b>1.3</b> | <b>Metodologia</b>                                             | <b>13</b> |
| 1.3.1      | Desenvolvimento do Ambiente Virtual                            | 13        |
| 1.3.2      | Implementação do Algoritmo Hybrid A*                           | 13        |
| 1.3.3      | Validação e Análise dos Resultados                             | 14        |
| <b>1.4</b> | <b>Organização</b>                                             | <b>14</b> |
| <b>2</b>   | <b>FUNDAMENTAÇÃO TEÓRICA</b>                                   | <b>15</b> |
| <b>2.1</b> | <b>Modelo Cinemático</b>                                       | <b>15</b> |
| <b>2.2</b> | <b>Grafos e Filas de Prioridade</b>                            | <b>17</b> |
| <b>2.3</b> | <b>Campo de Voronoi</b>                                        | <b>19</b> |
| <b>2.4</b> | <b>Algoritmo Brushfire</b>                                     | <b>22</b> |
| <b>2.5</b> | <b>Trajetória de Menor Comprimento e Curvas de Reeds-Shepp</b> | <b>27</b> |
| <b>2.6</b> | <b>Unity 3D como Ambiente de Simulação</b>                     | <b>28</b> |
| <b>3</b>   | <b>IMPLEMENTAÇÃO DO ALGORITMO HYBRID A*</b>                    | <b>32</b> |
| <b>3.1</b> | <b>Evolução dos Algoritmos de Busca</b>                        | <b>32</b> |
| 3.1.1      | Algoritmo de Dijkstra                                          | 33        |
| 3.1.2      | Heurística e Algoritmo A*                                      | 33        |
| 3.1.3      | Uso de Heurísticas no Planejamento de Trajetórias              | 34        |
| <b>3.2</b> | <b>Algoritmo Hybrid A*</b>                                     | <b>36</b> |
| 3.2.1      | Introdução ao Algoritmo Hybrid A*                              | 36        |
| 3.2.2      | Funcionamento do Algoritmo Hybrid A*                           | 36        |
| 3.2.2.1    | Expansão de Nós no Hybrid A*                                   | 37        |
| 3.2.2.2    | Expansão Analítica usando as Curvas de Reeds-Shepp             | 39        |
| 3.2.2.3    | Campo de Voronoi                                               | 39        |
| 3.2.2.4    | Otimização Local e Suavização                                  | 40        |
| 3.2.3      | Descrição da Execução do Hybrid A*                             | 41        |
| <b>4</b>   | <b>AMBIENTE VIRTUAL E INTEGRAÇÃO DO ALGORITMO HYBRID A*</b>    | <b>46</b> |
| <b>4.1</b> | <b>Dimensões do Veículo e Cálculo de Movimento</b>             | <b>46</b> |
| <b>4.2</b> | <b>Divisão dos <i>Scripts</i> e Cenários de Simulação</b>      | <b>48</b> |
| <b>4.3</b> | <b>Discretização e Detecção de Obstáculos</b>                  | <b>49</b> |
| <b>4.4</b> | <b>Heurísticas Holonômicas</b>                                 | <b>51</b> |

|       |                                             |     |
|-------|---------------------------------------------|-----|
| 4.4.1 | Campo de Fluxo . . . . .                    | 52  |
| 4.4.2 | A* . . . . .                                | 54  |
| 4.5   | Campo de Voronoi . . . . .                  | 56  |
| 4.6   | Hybrid A* . . . . .                         | 58  |
| 5     | CONCLUSÃO . . . . .                         | 66  |
|       | REFERÊNCIAS BIBLIOGRÁFICAS . . . . .        | 67  |
|       | ANEXO A – CLASSE GRID . . . . .             | 70  |
|       | ANEXO B – CLASSE CELL . . . . .             | 72  |
|       | ANEXO C – CLASSE MOVEMENT . . . . .         | 73  |
|       | ANEXO D – CLASSE CARMODEL . . . . .         | 74  |
|       | ANEXO E – CLASSE PRIORITYQUEUE . . . . .    | 75  |
|       | ANEXO F – CLASSE ASTARCELL . . . . .        | 77  |
|       | ANEXO G – CLASSE ASTAR . . . . .            | 78  |
|       | ANEXO H – CLASSE FLOWFIELDCELL . . . . .    | 80  |
|       | ANEXO I – CLASSE FLOWFIELD . . . . .        | 81  |
|       | ANEXO J – CLASSE VORONOIFIELDCELL . . . . . | 83  |
|       | ANEXO K – CLASSE VORONOIFIELD . . . . .     | 85  |
|       | ANEXO L – CLASSE REEDSSHEPP . . . . .       | 89  |
|       | ANEXO M – CLASSE NODE . . . . .             | 96  |
|       | ANEXO N – CLASSE HYBRIDASTAR . . . . .      | 97  |
|       | ANEXO O – CLASSE PATHBUILDER . . . . .      | 101 |

# 1 Introdução

Este trabalho propõe uma implementação detalhada do algoritmo Hybrid A\* aplicado ao planejamento de trajetórias em veículos autônomos, utilizando a *game engine* Unity 3D, como ambiente de simulação virtual. A pesquisa explora o potencial do Unity para criar simulações realistas, permitindo avaliar a eficácia e a precisão do algoritmo em condições diversas e complexas.

## 1.1 Contextualização

Nas últimas décadas, a área de veículos autônomos tem experimentado avanços notáveis, impulsionados pela evolução tecnológica e pela crescente demanda por soluções que promovam a segurança e a eficiência no transporte. Empresas pioneiras, como a Tesla, com seu sistema *Autopilot*, introduziram ao mercado veículos com capacidades autônomas de nível 2, de acordo com a classificação da *Society of Automotive Engineers* (SAE), onde o veículo pode controlar direção e aceleração/frenagem em determinadas condições, mas ainda requer supervisão do motorista (COMMITTEE, 2021). Essa tecnologia tem o potencial de transformar o setor de transporte, com impactos que abrangem o tráfego urbano, o transporte público e até setores como o agronegócio e a logística.

O impacto positivo dos veículos autônomos está intimamente ligado à segurança no trânsito. Estima-se que mais de 90% dos acidentes de trânsito resultam de falhas humanas, como distração e cansaço, e a automação veicular surge como uma solução promissora para mitigar drasticamente essas ocorrências (SINGH, 2015). Além disso, o uso eficiente da infraestrutura viária poderia otimizar o fluxo de veículos, reduzindo engarrafamentos e o tempo de viagem, trazendo benefícios para toda a sociedade.

Entre os componentes essenciais do desenvolvimento de veículos autônomos, o planejamento de trajetórias se destaca como um dos pilares centrais, pois envolve a geração de rotas seguras e otimizadas que respeitam as condições dinâmicas do ambiente e as limitações cinemáticas do veículo (PADEN et al., 2016). O algoritmo de planejamento de trajetórias precisa, portanto, lidar com situações complexas, como cruzamentos com tráfego intenso e mudanças inesperadas no ambiente, incluindo obras e acidentes.

A simulação computacional tem sido uma ferramenta valiosa no desenvolvimento e teste de algoritmos para veículos autônomos, permitindo que testes sejam realizados em cenários controlados, sem riscos à segurança física. A plataforma Unity (UNITY-TECHNOLOGIES, 2024), amplamente utilizada no desenvolvimento de jogos e simulações 3D, proporciona um ambiente virtual sofisticado para o treinamento e teste de algoritmos

de veículos autônomos (JULIANI, 2018). Com funcionalidades que incluem a criação de cenários complexos, sensores virtuais e capacidades de integração, o Unity possibilita simulações em larga escala que replicam a complexidade do mundo real, acelerando o desenvolvimento de sistemas autônomos e permitindo testes antes da implementação em veículos reais.

Neste contexto, a implementação e validação de algoritmos de planejamento de trajetórias tornam-se fundamentais para assegurar a precisão e a segurança da navegação autônoma. Este trabalho propõe o uso do algoritmo Hybrid A\*, uma extensão do tradicional algoritmo A\*, projetado para lidar com restrições não-holonômicas, cruciais em cenários de navegação de veículos (MONTEMERLO et al., 2008).

O algoritmo Hybrid A\* foi desenvolvido para superar as limitações do A\* clássico em problemas de navegação envolvendo veículos, que apresentam restrições específicas de direção e não podem realizar mudanças abruptas de trajetória ou movimentos laterais. Enquanto o A\* tradicional é eficaz para problemas de busca genéricos, o Hybrid A\* incorpora a cinemática do veículo, permitindo a geração de trajetórias suaves e fisicamente viáveis.

Neste trabalho, a escolha do Hybrid A\* justifica-se pela sua capacidade de planejar trajetórias eficientes e seguras em ambientes com obstáculos estáticos, respeitando as limitações físicas do veículo. Estudos como o de González et al. (2015) destacam o Hybrid A\* como uma técnica eficaz para o planejamento de trajetórias em veículos automatizados, especialmente em ambientes confinados e com obstáculos. Essa característica faz do Hybrid A\* uma opção ideal para simulações no Unity, onde a precisão e segurança na navegação podem ser rigorosamente avaliadas em um ambiente virtual controlado.

## 1.2 Objetivos

O objetivo geral deste trabalho é avaliar uma implementação do algoritmo Hybrid A\* desenvolvida na plataforma Unity, com foco no planejamento de trajetórias de veículos autônomos em ambiente estático e com obstáculos. Para alcançar esse objetivo geral, o trabalho se desdobra nos seguintes objetivos específicos:

- implementar uma versão do algoritmo Hybrid A\*, utilizando a plataforma Unity, visando garantir o desempenho e a confiabilidade do algoritmo em tempo de execução.
- criar um ambiente de simulação realista no Unity, utilizando suas capacidades de modelagem 3D e simulação física, para avaliar o desempenho do algoritmo em cenários e ambiente semelhantes aos casos reais. Este caso visa contemplar a presença de obstáculos estáticos, tais como vias, estacionamentos, outros veículos estacionados etc.

- realizar uma análise detalhada do comportamento do algoritmo em cenários variados de ambiente restrito, observando a precisão no planejamento de trajetórias e capacidade de lidar com as restrições cinemáticas do veículo.
- desenvolver ferramentas de visualização de dados que permitam acompanhar o processo de planejamento de trajetórias em tempo real, facilitando a compreensão das decisões tomadas pelo algoritmo e a identificação de potenciais melhorias na navegação em ambientes de espaço limitado.
- documentar a implementação de forma modular e legível, promovendo uma estrutura de código que permita futuras adaptações e a replicação do algoritmo para outros tipos de simulação.

Ao cumprir esses objetivos, o trabalho pretende contribuir para o avanço da pesquisa em veículos autônomos, destacando o uso da simulação computacional e da plataforma Unity como ferramentas valiosas para o teste e o aprimoramento de algoritmos de planejamento de trajetórias em ambientes restritos.

## 1.3 Metodologia

Nesta seção, descreve-se a metodologia adotada para a implementação e validação do algoritmo Hybrid A\* no planejamento de trajetórias de veículos autônomos. A metodologia divide-se em três etapas principais: (i) desenvolvimento do ambiente virtual de simulação utilizando a plataforma Unity 3D, (ii) implementação do algoritmo Hybrid A\* em C# para o planejamento de trajetórias e (iii) validação e análise dos resultados obtidos em diferentes cenários de simulação.

### 1.3.1 Desenvolvimento do Ambiente Virtual

Para a criação de um ambiente de simulação adequado, utiliza-se a plataforma Unity 3D, amplamente reconhecida pela capacidade de simular ambientes tridimensionais e interativos com alta fidelidade. No Unity, modelam-se cenários que representam ambientes típicos de navegação autônoma, incluindo obstáculos estáticos que impõem desafios ao planejamento de rotas. A interface do Unity, com ferramentas como *GameObjects*, *scripts* em C# e hierarquias organizacionais, facilita o desenvolvimento modular da simulação, possibilitando uma implementação organizada e escalável.

### 1.3.2 Implementação do Algoritmo Hybrid A\*

A implementação do algoritmo Hybrid A\* foi realizada em C# devido à integração nativa da linguagem com o Unity. O código foi estruturado de forma a garantir modularidade

e facilidade de adaptação, dividindo-se em classes que representam as funções essenciais para o planejamento de trajetória, como a geração de nós e a aplicação de heurísticas.

### 1.3.3 Validação e Análise dos Resultados

Para validar a eficácia do algoritmo, realizam-se testes em diferentes cenários, variando a quantidade e a disposição dos obstáculos. A análise dos resultados enfoca a precisão e a eficiência do algoritmo em traçar trajetórias viáveis e seguras, observando-se a capacidade do Hybrid A\* de lidar com restrições não-holonômicas e a adequação das trajetórias geradas para veículos autônomos.

## 1.4 Organização

Este trabalho está estruturado em cinco capítulos, cada um abordando uma etapa essencial para o desenvolvimento e análise do algoritmo Hybrid A\* aplicado ao planejamento de trajetórias de veículos autônomos em ambiente virtual.

No **Capítulo 1**, *Introdução*, contextualiza-se o tema e apresentam-se os objetivos do trabalho, com destaque para a relevância do uso de simulação virtual no desenvolvimento de algoritmos de planejamento de trajetórias para veículos autônomos.

O **Capítulo 2**, *Fundamentação Teórica*, aborda os conceitos necessários para a compreensão do modelo cinemático, algoritmos de planejamento de trajetória e as ferramentas utilizadas, como o Unity, que servem de base para a implementação do algoritmo Hybrid A\*.

Em **Capítulo 3**, *Implementação do Algoritmo Hybrid A\**, apresenta-se detalhadamente o algoritmo Hybrid A\*, incluindo as heurísticas empregadas e o processo de expansão de nós, discutindo-se as particularidades que diferem o Hybrid A\* do algoritmo A\* tradicional.

O **Capítulo 4**, *Ambiente Virtual e Integração do Algoritmo Hybrid A\**, descreve-se o processo de desenvolvimento do ambiente virtual no Unity e a implementação do algoritmo em C#, além de apresentar os resultados da simulação, com foco na análise da eficácia do planejamento de trajetórias em diferentes cenários.

Por fim, no **Capítulo 5**, *Conclusão*, apresentam-se as considerações finais, as contribuições do trabalho e sugestões para futuras pesquisas que possam expandir ou aprimorar o uso do algoritmo Hybrid A\* em sistemas de navegação autônoma.

## 2 Fundamentação Teórica

Neste capítulo, são apresentados os conceitos fundamentais e as teorias essenciais que sustentam o desenvolvimento e a implementação do algoritmo Hybrid A\* para o planejamento de trajetórias de veículos autônomos em ambientes virtuais. Primeiramente, aborda-se o modelo cinemático que representa a dinâmica e as restrições físicas dos veículos, estabelecendo uma base para a compreensão das limitações de movimento e manobrabilidade. Em seguida, explora-se tópicos fundamentais de teoria de grafos e filas de prioridade, essenciais para o entendimento dos algoritmos de busca, como o A\* e suas variações, no contexto do planejamento de trajetórias.

Além disso, discutem-se o Diagrama de Voronoi e o Algoritmo Brushfire, técnicas matemáticas e computacionais utilizadas para a criação de mapas de influência e áreas de segurança ao redor de obstáculos. Esses conceitos são cruciais para a geração de trajetórias seguras em ambientes com obstáculos. Outro ponto abordado são as curvas de Reeds-Shepp, que definem trajetórias de menor comprimento considerando restrições de reversão e curvatura mínima, ampliando a eficiência na navegação de veículos em espaços confinados.

Por fim, apresenta-se o Unity como o ambiente de simulação escolhido para implementar e testar o algoritmo Hybrid A\*, detalhando sua interface, organização de projeto e ferramentas essenciais, como *GameObjects* e *scripts* em C#.

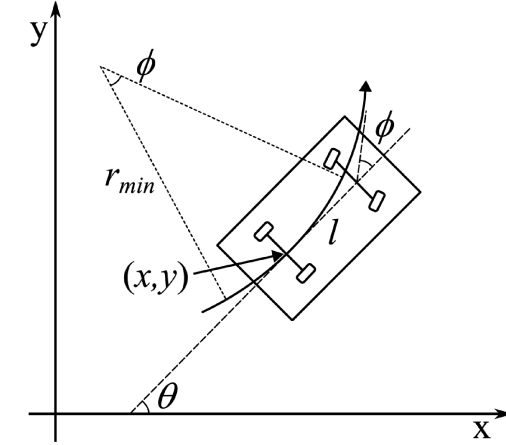
### 2.1 Modelo Cinemático

Ao implementar algoritmos de planejamento de trajetórias para veículos autônomos, a escolha do modelo de veículo é fundamental para capturar as limitações físicas e cinemáticas que influenciam a viabilidade e precisão das trajetórias geradas. Neste trabalho, adota-se o *bicycle model*, amplamente utilizado para representar veículos com configuração similar a automóveis devido à sua simplicidade e aderência às restrições não-holonômicas (LAVALLE, 2006). Esse modelo representa o veículo como um ponto localizado no eixo traseiro, considerando que ele se move a uma velocidade constante em um plano bidimensional.

O *bicycle model* considera que o veículo segue trajetórias contínuas sem possibilidade de deslizamento lateral, devido à restrição não-holonômica, isto é, o veículo só se move na direção para onde está apontado, sendo incapaz de realizar deslocamentos laterais. Esse modelo também impõe um raio mínimo de curvatura, determinado pela configuração geométrica do veículo e pelo ângulo máximo de esterçamento das rodas dianteiras. Esse raio mínimo é um fator essencial no planejamento de trajetórias, pois define a curvatura

mínima que o veículo pode descrever em movimento, restringindo as manobras possíveis em espaços confinados. A Figura 1 ilustra os parâmetros elementares utilizados para representar o modelo cinemático deste veículo.

Figura 1 – Modelo cinemático de um veículo do tipo carro, representado pelo modelo de bicicleta.



Fonte: Adaptado de Xu et al. (2011).

Matematicamente, o estado do veículo é descrito por três variáveis: a posição no plano  $(x, y)$  e a orientação  $\theta$ , que representa o ângulo da direção do veículo em relação a um referencial fixo. A cinemática do modelo é governada pelas seguintes equações diferenciais (POPP et al., 2010):

$$\dot{x} = v \cos(\theta), \quad (2.1)$$

$$\dot{y} = v \sin(\theta), \quad (2.2)$$

$$\dot{\theta} = \frac{v}{l} \tan(\phi), \quad (2.3)$$

onde:

- $v$  é a velocidade linear constante do veículo;
- $l$  é a distância entre os eixos do veículo (entre as rodas traseiras e dianteiras);
- $\phi$  é o ângulo de direção do veículo (ângulo de esterçamento das rodas dianteiras).

A restrição de não deslizamento lateral pode ser expressa pela seguinte condição:

$$\dot{y} \cos(\theta) - \dot{x} \sin(\theta) = 0, \quad (2.4)$$

indicando que o movimento lateral (perpendicular à direção de movimento) é inexistente, reforçando que o veículo só pode se mover para frente ou para trás, conforme a direção em que está apontado. Além disso, o raio mínimo de curvatura  $r_{\min}$  é dado por:

$$r_{\min} = \frac{l}{\tan(\phi_{\max})}, \quad (2.5)$$

onde  $\phi_{\max}$  é o ângulo máximo de esterçamento das rodas dianteiras. Essa limitação física deve ser considerada ao calcular a trajetória ideal do ponto inicial até o objetivo, pois restringe a curvatura das trajetórias viáveis.

Para o cálculo da trajetória, a posição da roda traseira é representada por  $(x, y)$  e o ângulo de orientação do veículo é dado por  $\theta$  (em radianos). Dependendo do tipo de movimento especificado, as equações para determinar a nova posição e orientação do veículo variam:

1. Movimento com Curva: Quando o veículo executa uma curva, o movimento é calculado com base no raio mínimo de curvatura  $r_{\min}$ . O centro da curva,  $(x_c, y_c)$ , é determinado a partir do raio e da orientação inicial do veículo. Após um deslocamento  $d$ , o novo ângulo  $\theta'$  é dado por:

$$\theta' = \theta + \frac{d}{2\pi r_{\min}} \quad (2.6)$$

A nova posição  $(x', y')$  do veículo é então obtida pela rotação em torno do centro da curva:

$$x' = x_c + r_{\min} \cos(\theta') \quad (2.7)$$

$$y' = y_c + r_{\min} \sin(\theta') \quad (2.8)$$

2. Movimento em Linha Reta: Quando o veículo se desloca em linha reta, a nova posição é calculada diretamente com base no deslocamento  $d$ :

$$x' = x + d \cos(\theta) \quad (2.9)$$

$$y' = y + d \sin(\theta) \quad (2.10)$$

## 2.2 Grafos e Filas de Prioridade

A teoria dos grafos é fundamental no campo da ciência da computação, especialmente para a resolução de problemas que envolvem a representação de relações e interações entre diferentes elementos. Um grafo  $G = (V, E)$  é definido por um conjunto de vértices  $V$ , que representam os elementos ou estados, e por um conjunto de arestas  $E$ , que conecta pares de vértices e simboliza as relações ou transições entre esses elementos (CORMEN et al., 2022). Em problemas de planejamento de trajetórias, os vértices de um grafo podem representar diferentes estados do veículo ou posições no ambiente, enquanto as arestas indicam as possíveis transições entre esses estados. O custo de cada aresta, muitas vezes calculado com base na distância, no tempo ou no consumo de energia, permite quantificar a “dificuldade” ou o “esforço” associado a cada transição.

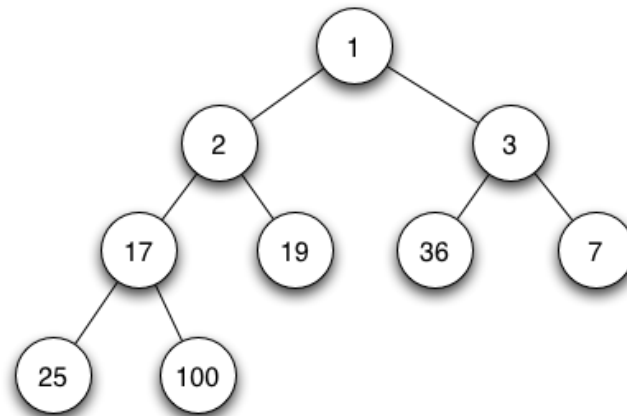
Grafos podem ser classificados em direcionados ou não direcionados, dependendo se as arestas possuem uma orientação específica ou não. Em grafos direcionados, cada

aresta possui uma direção, o que significa que o caminho de um vértice para outro é unidirecional, sendo possível que um vértice  $v$  tenha uma conexão com  $u$  mas não vice-versa. Em contrapartida, em grafos não direcionados, as arestas conectam os vértices de forma bidirecional, sem imposição de sentido. Os grafos também podem ser ponderados, quando cada aresta possui um valor associado que representa o custo para se mover entre dois vértices adjacentes. Grafos ponderados são amplamente utilizados no planejamento de trajetórias e em problemas de navegação, onde o custo das arestas representa a distância ou a dificuldade em percorrer o caminho de um ponto a outro.

A busca por um caminho de menor custo em um grafo ponderado é um problema essencial em diversos algoritmos de planejamento de trajetórias, especialmente em sistemas de navegação autônoma. Em algoritmos como o Dijkstra e o  $A^*$ , o objetivo é encontrar a rota mais eficiente, minimizando o custo total entre um vértice inicial e um vértice de destino. Para otimizar a exploração dos vértices e reduzir o tempo de execução, esses algoritmos fazem uso de uma estrutura de dados chamada fila de prioridade, que armazena os vértices de acordo com seu custo acumulado, garantindo que o próximo vértice a ser processado seja sempre aquele com o menor custo até o momento.

Uma fila de prioridade é uma estrutura de dados que permite a inserção de elementos com diferentes valores de prioridade, onde o elemento com a menor (ou maior) prioridade é extraído primeiro. No contexto de algoritmos de busca em grafos ponderados, a fila de prioridade é essencial, pois possibilita o processamento dos vértices na ordem que minimiza o custo acumulado, otimizando o desempenho do algoritmo. Em algoritmos de busca em grafos ponderados, uma fila de prioridade assegura que o vértice com o menor custo acumulado seja explorado antes, reduzindo o número de vértices a serem considerados e acelerando a busca pela solução.

Uma implementação eficiente de fila de prioridade é a estrutura de dados chamada *binary heap*, ou heap binário. A *binary heap* é uma árvore binária completa onde cada nó pai possui uma chave menor (no caso de um *min-heap*, a exemplo da Figura 2) ou maior (no caso de um *max-heap*) que as chaves de seus filhos diretos. Essa propriedade de heap permite que o elemento de menor prioridade seja sempre mantido na raiz, ou seja, na primeira posição da árvore. A principal vantagem da *binary heap* é que ele possibilita a inserção e a remoção de elementos com complexidade  $O(\log n)$ , sendo altamente eficiente para a implementação de filas de prioridade em algoritmos de busca (KNUTH, 1997).

Figura 2 – Exemplo de *min-heap* para organização de prioridade.

Fonte: Onar Vikingstad, 2005. Disponível em:

<https://commons.wikimedia.org/wiki/File:Min-heap.png>

Em algoritmos de busca em grafos, a *binary heap* é utilizado para organizar os vértices conforme seu custo acumulado, facilitando a extração do vértice com o menor custo a cada iteração do algoritmo. A *binary heap* armazena os vértices de maneira compacta, utilizando um *array* para representar a estrutura de árvore completa, onde cada posição do *array* corresponde a um nó da árvore. Essa organização permite operações eficientes de inserção e extração, que são cruciais para manter a performance do algoritmo de busca em tempo real. A inserção de um novo vértice na fila de prioridade envolve a operação de subida (ou *heapify-up*), enquanto a extração do vértice de menor prioridade, localizado na raiz, utiliza a operação de descida (*heapify-down*) para restaurar a propriedade de heap após a remoção.

Assim, o uso de uma fila de prioridade implementada com *binary heap* em algoritmos de busca garante que o algoritmo processe os vértices de maneira ordenada e eficiente, minimizando o custo total do caminho explorado e o tempo de execução. Dessa forma, a *binary heap* não apenas melhora o desempenho dos algoritmos de busca, mas também contribui para a viabilidade prática da navegação autônoma.

## 2.3 Campo de Voronoi

O Diagrama de Voronoi é uma estrutura matemática que divide um espaço em regiões, chamadas de células de Voronoi, com base em um conjunto de pontos geradores. Dado um conjunto de pontos  $P = \{p_1, p_2, \dots, p_n\}$  em um plano, cada ponto  $p_i \in P$  é associado a uma célula de Voronoi  $V(p_i)$ , que consiste em todos os pontos do plano que estão mais próximos de  $p_i$  do que de qualquer outro ponto  $p_j \neq p_i$  (AURENHAMMER,

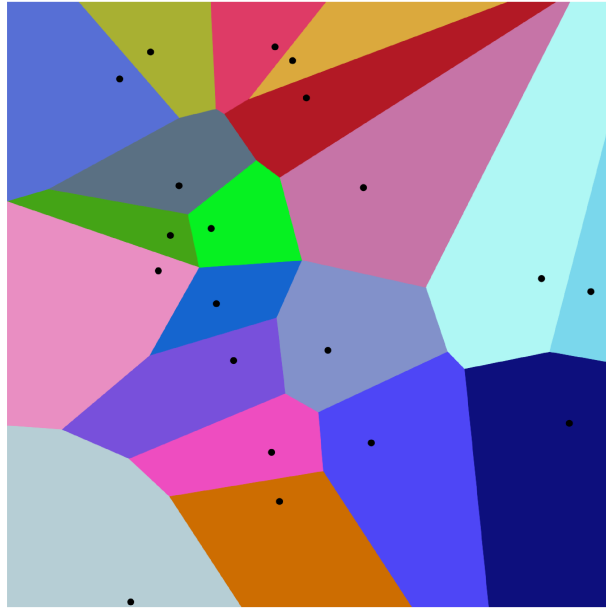
1991). Formalmente, a célula de Voronoi  $V(p_i)$  de um ponto  $p_i$  é definida como:

$$V(p_i) = \{x \in \mathbb{R}^2 \mid \forall p_j \in P, \text{dist}(x, p_i) \leq \text{dist}(x, p_j)\}, \quad (2.11)$$

onde  $\text{dist}(x, p_i)$  representa a distância euclidiana entre o ponto  $x$  e o ponto gerador  $p_i$ . A divisão resultante cria uma tesselação do plano em regiões convexas, onde cada célula é limitada por arestas formadas por segmentos de linha reta que são perpendiculares às linhas que conectam os pontos geradores adjacentes. Essas fronteiras são chamadas de bissetores e representam o lugar geométrico dos pontos equidistantes entre dois pontos geradores vizinhos.

A Figura 3 ilustra um exemplo de Diagrama de Voronoi para um conjunto de pontos geradores. Cada região colorida corresponde a uma célula de Voronoi associada a um ponto gerador, demonstrando como o espaço é particionado de acordo com a proximidade a esses pontos.

Figura 3 – Exemplo de um Diagrama de Voronoi gerado a partir de um conjunto de pontos.



Fonte: Balu Ertl, 2015. Disponível em:

<[https://commons.wikimedia.org/wiki/File:Euclidean\\_Voronoi\\_diagram.svg](https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg)>

O Diagrama de Voronoi tradicional utiliza pontos discretos como geradores, criando células baseadas na distância euclidiana para cada ponto do conjunto. Nesse contexto, cada ponto no espaço pertence a uma célula específica, delimitada pela proximidade a um gerador em comparação aos demais. No entanto, o Diagrama Generalizado de Voronoi (GVD) expande esse conceito ao considerar geradores que não são apenas pontos isolados, mas também formas geométricas mais complexas, como segmentos de linha, curvas e polígonos. Assim, no GVD, a região associada a cada gerador é composta por todos os pontos do plano que estão mais próximos desse gerador do que de qualquer outro, levando em conta a menor distância ao contorno dos objetos geométricos.

A principal vantagem do GVD é a sua capacidade de modelar ambientes mais complexos e realistas, especialmente em aplicações que requerem uma representação detalhada do espaço para fins de planejamento de movimento. Quando um segmento de linha, por exemplo, é utilizado como gerador, a célula de Voronoi correspondente é definida pelo conjunto de pontos cuja proximidade é maior a qualquer ponto do segmento do que a outros geradores. Esse nível de detalhamento possibilita a criação de fronteiras entre regiões compostas por arcos e segmentos, que refletem a geometria dos geradores de maneira mais precisa, contrastando com os bissetores lineares observados no Diagrama de Voronoi tradicional.

Essa abordagem é especialmente útil em planejamento de movimento, onde obstáculos de formas complexas precisam ser representados de forma detalhada para garantir a segurança e eficiência das trajetórias calculadas. [Takahashi e Schilling \(1989\)](#) demonstram a aplicabilidade do GVD para o planejamento de movimento, destacando como a divisão do espaço com base na proximidade geométrica permite evitar colisões e otimizar a trajetória ao navegar em ambientes com obstáculos variados.

No planejamento de trajetórias para veículos autônomos, o Diagrama de Voronoi e o Diagrama Generalizado de Voronoi (GVD) desempenham um papel essencial ao estabelecer áreas de influência ao redor dos obstáculos e orientar o movimento do veículo em regiões que maximizam a distância de segurança em relação a esses obstáculos. Esses diagramas criam uma divisão espacial que permite ao veículo identificar as zonas de proximidade de forma precisa, facilitando a seleção de caminhos seguros e otimizados. Ao estruturar o espaço em células definidas pela proximidade a obstáculos, o GVD, em particular, oferece um mapeamento detalhado e adaptável às formas dos obstáculos, contribuindo para trajetórias que respeitam as complexidades do ambiente.

Para integrar essa informação ao processo de planejamento de trajetórias, utiliza-se o Campo de Voronoi como critério orientador ([DOLGOV et al., 2008](#)). O Campo de Voronoi é uma função de custo que atribui valores a cada ponto do espaço, ajustando-se dinamicamente à proximidade dos obstáculos. A formulação matemática dessa função de custo permite ponderar diversos fatores, resultando em trajetórias que equilibram segurança e eficiência.

A função de custo do Campo de Voronoi é expressa como:

$$\rho_V(x, y) = \left( \frac{\alpha}{\alpha + d_O(x, y)} \right) \left( \frac{d_V(x, y)}{d_O(x, y) + d_V(x, y)} \right) \left( \frac{(d_O - d_O^{\max})^2}{(d_O^{\max})^2} \right), \quad (2.12)$$

onde:

- $d_O(x, y)$  representa a distância do ponto  $(x, y)$  ao obstáculo mais próximo;
- $d_V(x, y)$  é a distância até a fronteira (borda) mais próxima;

- $d_O^{\max}$  corresponde à distância máxima considerada segura em relação aos obstáculos;
- $\alpha$  é um parâmetro de ajuste que controla a influência da proximidade aos obstáculos.

Essa função é projetada para favorecer trajetórias que maximizam a distância em relação aos obstáculos, preservando a segurança. A primeira parte da função assegura que o custo é elevado para pontos próximos a obstáculos, reduzindo-se à medida que essa distância aumenta. A segunda parte harmoniza a proximidade à borda com a distância ao obstáculo, enquanto a terceira parte penaliza pontos que excedem a distância máxima de segurança.

## 2.4 Algoritmo Brushfire

O algoritmo Brushfire é uma técnica de mapeamento e planejamento amplamente utilizada na robótica para representar o ambiente e auxiliar na navegação autônoma (CHOSSET et al., 2005). Seu funcionamento é baseado na propagação de ondas a partir de um conjunto inicial de células, preenchendo gradualmente o espaço com valores de distância que indicam o custo de movimentação a partir das origens da expansão, conforme o Algoritmo 1. Dependendo do ponto de partida da expansão, o algoritmo Brushfire pode ser utilizado tanto para atração quanto para repulsão, orientando a navegação do robô de acordo com o objetivo desejado.

O algoritmo Brushfire é especialmente útil para criar um campo de fluxo (*flow field*), que é uma representação discreta do espaço onde cada célula contém uma direção de movimento sugerida. Se o ponto inicial de expansão do Brushfire for um conjunto de obstáculos, o campo de fluxo gerado indicará direções que afastam o robô desses obstáculos, criando um campo de repulsão. Por outro lado, se a expansão começar a partir de um ponto de destino, o campo de fluxo indicará direções que guiam o robô até esse objetivo, criando um campo de atração. Essas duas abordagens são úteis para diferentes aspectos da navegação autônoma: enquanto o campo de repulsão ajuda a evitar colisões, o campo de atração guia o robô ao seu destino de forma eficiente.

Matematicamente, o campo de fluxo gerado pelo algoritmo Brushfire pode ser descrito como uma matriz em que cada célula  $(x, y)$  contém um vetor de direção  $\vec{d}(x, y)$  que aponta para a direção de menor custo ou menor risco. Durante a execução do algoritmo, cada célula é atribuída um valor que representa a sua distância ao ponto de origem da expansão. A direção do vetor em cada célula é determinada com base nos valores das células adjacentes, apontando para a célula com menor valor, indicando assim o fluxo desejado.

O campo de repulsão, a exemplo da Figura 4, é especialmente útil em ambientes onde há muitos obstáculos, pois cria uma zona de segurança ao redor deles. Cada célula próxima

Figura 4 – Exemplo de campo de fluxo gerado pelo algoritmo Brushfire para repulsão.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 |
| 2 | 3 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 2 | 1 | 2 |
| 2 | 3 | 2 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 4 | 3 | 2 | 1 | 1 |
| 2 | 2 | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 3 | 2 | 1 |
| 2 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 2 |
| 2 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 1 | 1 | 1 | 1 |
| 2 | 3 | 3 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 2 |
| 2 | 3 | 4 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 3 | 3 | 3 | 2 |
| 2 | 3 | 4 | 4 | 5 | 5 | 6 | 7 | 7 | 7 | 6 | 5 | 4 | 4 | 3 | 2 |
| 2 | 3 | 4 | 4 | 4 | 4 | 5 | 6 | 6 | 6 | 7 | 6 | 5 | 4 | 3 | 2 |
| 2 | 3 | 4 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 6 | 5 | 4 | 3 | 2 |
| 2 | 3 | 3 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 4 | 3 | 2 | 2 |
| 1 | 2 | 2 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 3 | 2 | 1 | 1 |
| 1 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 2 | 1 | 1 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |

Fonte: Gerado em: <<https://web.ist.utl.pt/~margaridaacferreira/RoboticsFun/>>

aos obstáculos é associada a um vetor que aponta para longe dos mesmos, garantindo que o robô mantenha uma distância segura enquanto se desloca. Esse método é frequentemente utilizado em ambientes dinâmicos, onde a posição dos obstáculos pode mudar, pois o campo de repulsão pode ser recalculado rapidamente para refletir novas configurações do ambiente.

Figura 5 – Exemplo de campo de fluxo gerado pelo algoritmo Brushfire para atração.

|    |   |   |   |    |    |    |    |
|----|---|---|---|----|----|----|----|
| 7  | 6 | 5 | 4 | 5  | 6  | 7  | 8  |
| 6  | 5 | 4 | 3 | 4  | 5  | 1  | 9  |
| 5  | 4 | 3 | 2 | 3  | 4  | 1  | 8  |
| 6  | 5 | 4 | 1 | 4  | 5  | 6  | 7  |
| 7  | 6 | 5 | 1 | 1  | 1  | 7  | 8  |
| 8  | 1 | 6 | 7 | 8  | 9  | 8  | 9  |
| 9  | 1 | 7 | 8 | 9  | 10 | 9  | 10 |
| 10 | 9 | 8 | 9 | 10 | 11 | 10 | 11 |

Fonte: Gerado em: <<https://web.ist.utl.pt/~margaridaacferreira/RoboticsFun/>>

Já o campo de atração, onde a expansão do Brushfire é iniciada no destino, orienta o movimento do robô de qualquer ponto inicial até o objetivo, minimizando o caminho até

o destino. Neste caso, cada célula aponta na direção que leva ao objetivo, formando um caminho gradiente descendente em direção ao destino, conforme a Figura 5.

Embora o algoritmo Brushfire seja eficaz em diversos cenários, ele possui limitações significativas que podem comprometer sua aplicabilidade em algumas situações. Uma das principais desvantagens é o problema de aprisionamento em mínimos locais, onde o robô ou veículo pode ficar preso em áreas entre obstáculos devido ao gradiente local. Esse fenômeno ocorre porque o campo de repulsão gerado pelo algoritmo Brushfire cria “vales” de potencial entre obstáculos. Nesses vales, todas as direções indicadas pelo campo conduzem a um ponto de equilíbrio local, o que impede o movimento do robô para fora dessa região confinada.

Esses mínimos locais representam zonas em que o campo de repulsão, ao invés de direcionar o robô para longe dos obstáculos, limita sua mobilidade, impossibilitando-o de alcançar o objetivo final. Para mitigar essa limitação, o algoritmo Brushfire pode ser combinado com algoritmos de busca, como o  $A^*$ , que são projetados para encontrar trajetórias ótimas mesmo em espaços de busca complexos. A integração com o  $A^*$  permite que o sistema escape das armadilhas de aprisionamento ao identificar e seguir uma rota alternativa que não está restrita pelos vales de potencial do campo de repulsão. Dessa forma, o robô ou veículo consegue superar os mínimos locais, navegando eficientemente em direção ao seu destino, mesmo em ambientes complexos e com múltiplos obstáculos.

No contexto do Diagrama Generalizado de Voronoi (GVD), o algoritmo Brushfire é utilizado para construir o GVD por meio da expansão de ondas a partir dos obstáculos presentes no ambiente, conforme o Algoritmo 2. Nesse caso, o algoritmo gera um campo de repulsão onde cada célula recebe um valor correspondente à sua distância ao obstáculo mais próximo. As células que estão equidistantes de dois ou mais obstáculos formam o esqueleto central do GVD, que representa o caminho mais seguro e distante dos obstáculos.

Para a construção do GVD, o Brushfire inicia a expansão de todas as células que correspondem a obstáculos no ambiente, marcando-as como fontes iniciais. À medida que a expansão progride, cada célula adjacente recebe um valor incremental que indica a sua distância ao obstáculo mais próximo. O processo continua até que todas as células do ambiente estejam preenchidas com a menor distância aos obstáculos. O GVD é então formado pelas células que possuem múltiplos vizinhos com o mesmo valor de distância, indicando que essas células estão equidistantes de dois ou mais obstáculos.

**Algoritmo 1:** Pseudocódigo para o Brushfire.

```

1 Discretize o ambiente e obtenha a grade com obstáculos identificados;
2 para cada célula ( $i, j$ ) na grade faça
3   | Reinicie o estado da célula na posição ( $i, j$ );
4   | Defina os vizinhos da célula;
5 fim
6 Crie um conjunto aberto (fila) para rastrear as células a serem processadas;
7 Crie um conjunto fechado para registrar as células já processadas;
8 para cada célula inicial (obstáculos ou objetivo) faça
9   | Defina o custo da célula como 0;
10  | Adicione a célula ao conjunto aberto e ao conjunto fechado;
11 fim
12 enquanto conjunto aberto não estiver vazio faça
13   | Retire uma célula do conjunto aberto;
14   para cada vizinho da célula atual faça
15     | Calcule a distância entre a célula atual e o vizinho;
16     | Calcule o novo custo como o custo da célula atual mais essa distância;
17     se novo custo > custo do vizinho então
18       | Continue para o próximo vizinho;
19     fim
20     | Atualize o custo do vizinho para o novo custo;
21     se vizinho já estiver no conjunto fechado então
22       | Continue para o próximo vizinho;
23     fim
24     | Adicione o vizinho ao conjunto aberto e fechado;
25   fim
26 fim

```

**Algoritmo 2:** Pseudocódigo para o cálculo do Campo de Voronoi.

```

1  Discretize o ambiente e obtenha a grade com obstáculos identificados;
2  para cada célula  $(i, j)$  na grade faça
3      se a célula está obstruída e sua região ainda não foi preenchida então
4          Defina a região do obstáculo usando preenchimento de área;
5      fim
6  fim
7  Inicialize uma grade auxiliar sem obstáculos para o cálculo do campo de fluxo;
8  para cada célula  $(i, j)$  na grade faça
9      Copie a região da célula  $(i, j)$  da grade principal para a célula  $(i, j)$  da grade
        auxiliar;
10 fim
11 Crie um conjunto de células para usar no cálculo do campo de fluxo;
12 para cada célula  $(i, j)$  na grade faça
13     se a célula  $(i, j)$  está obstruída então
14         Adicione a célula  $(i, j)$  ao conjunto;
15     fim
16 fim
17 Calcule o campo de fluxo na grade auxiliar usando o conjunto de células dos
    obstáculos;
18 para cada célula  $(i, j)$  na grade faça
19     Defina a distância ao obstáculo mais próximo na célula  $(i, j)$  da grade principal
        como o valor do campo de fluxo da célula  $(i, j)$  da grade auxiliar;
20     Defina a região da célula  $(i, j)$  da grade principal com base na região da célula
         $(i, j)$  da grade auxiliar;
21 fim
22 para cada célula  $(i, j)$  na grade faça
23     para cada célula vizinha faça
24         se a célula vizinha pertence a uma região diferente então
25             Marque a célula atual como uma borda de região;
26             Continue para a próxima célula;
27         fim
28     fim
29 fim
30 Inicialize uma nova grade auxiliar sem obstáculos para o cálculo de campo de fluxo;
31 Crie um novo conjunto de células para o cálculo do campo de fluxo;
32 para cada célula  $(i, j)$  na grade faça
33     Adicione a célula  $(i, j)$  marcada como borda ao conjunto;
34 fim
35 Calcule o campo de fluxo na nova grade auxiliar usando o conjunto de células das
    bordas;
36 para cada célula  $(i, j)$  na grade faça
37     Defina a distância à borda mais próxima na célula  $(i, j)$  da grade principal
        como o valor do campo de fluxo da célula  $(i, j)$  da grade auxiliar;
38     Calcule o valor  $\rho_V$  da célula  $(i, j)$  usando a distância ao obstáculo e à borda;
39 fim

```

## 2.5 Trajetória de Menor Comprimento e Curvas de Reeds-Shepp

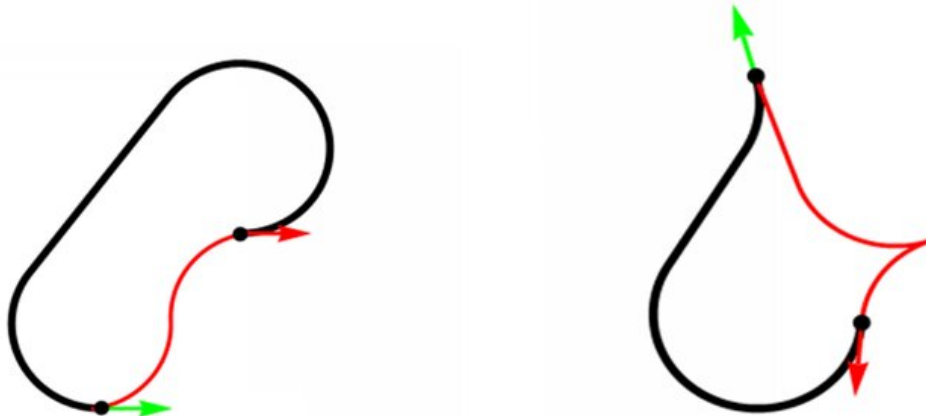
As curvas de [Reeds e Shepp \(1990\)](#) definem as trajetórias de menor comprimento para veículos que podem se mover tanto para frente quanto para trás, com restrições de raio mínimo de curva. Esse problema é uma extensão das curvas de [Dubins \(1957\)](#), que limitam o movimento a uma única direção, ou seja, sem a possibilidade de marcha à ré. As curvas de Dubins incluem apenas seis combinações de segmentos curvos e retos (CSC e CCC), pois são suficientes para garantir o menor trajeto entre dois pontos, sem permitir reversão. Já as curvas de Reeds-Shepp ampliam essa possibilidade, introduzindo um conjunto mais complexo de trajetórias, que incorporam mudanças de direção e segmentos reversos.

Reeds e Shepp demonstraram que, para um veículo com capacidade de reversão, é necessário considerar até 48 sequências de trajetórias para obter o caminho mais curto, o que é uma expansão considerável em relação às seis combinações de Dubins. Essas 48 sequências são chamadas de “palavras-base” e representam todas as possíveis combinações de segmentos circulares e retos, com ou sem reversão. No entanto, estudos subsequentes, como os de [Sussmann e Tang \(1991\)](#), mostraram que duas dessas sequências são redundantes, reduzindo o conjunto final para 46 trajetórias essenciais.

A notação das trajetórias de Reeds-Shepp utiliza os símbolos  $C$  e  $S$  para representar segmentos curvos e retos, respectivamente, além de um símbolo especial,  $|$ , que indica uma troca de direção entre os segmentos (de marcha à ré para frente ou vice-versa). Por exemplo, uma sequência  $C|C|C$  indica que o veículo realiza uma curva em marcha à ré, seguida de outra curva em movimento para frente, e assim sucessivamente. Em algumas combinações, utiliza-se um subíndice  $\pi/2$ , indicando que a curva deve ser seguida por um ângulo específico (como  $\pi/2$  radianos), para garantir a precisão do percurso.

Na Figura 6 é possível observar exemplos de trajetórias usando as curvas de Reeds-Shepp em comparação com as de Dubins. A flexibilidade oferecida pela reversão permite ao veículo alcançar trajetos mais curtos e manobrar de maneira mais eficiente em espaços confinados.

Figura 6 – Exemplo de trajetórias baseadas nas curvas de Reeds-Shepp e Dubins.



Fonte: [Sviatov et al. \(2023\)](#)

Para construir uma trajetória usando as 46 palavras-base das curvas de Reeds-Shepp, deve-se considerar que a sequência máxima de segmentos é composta por cinco primitivas, onde cada primitiva pode ser uma curva à esquerda ( $L$ ), uma curva à direita ( $R$ ) ou um segmento reto ( $S$ ). Além disso, cada uma dessas primitivas pode ter uma direção para frente (+) ou para trás (−), resultando em uma notação do tipo  $L^+S^-R^+$ , que indica uma curva à esquerda para frente, seguida de um segmento reto em marcha à ré, e uma curva à direita para frente.

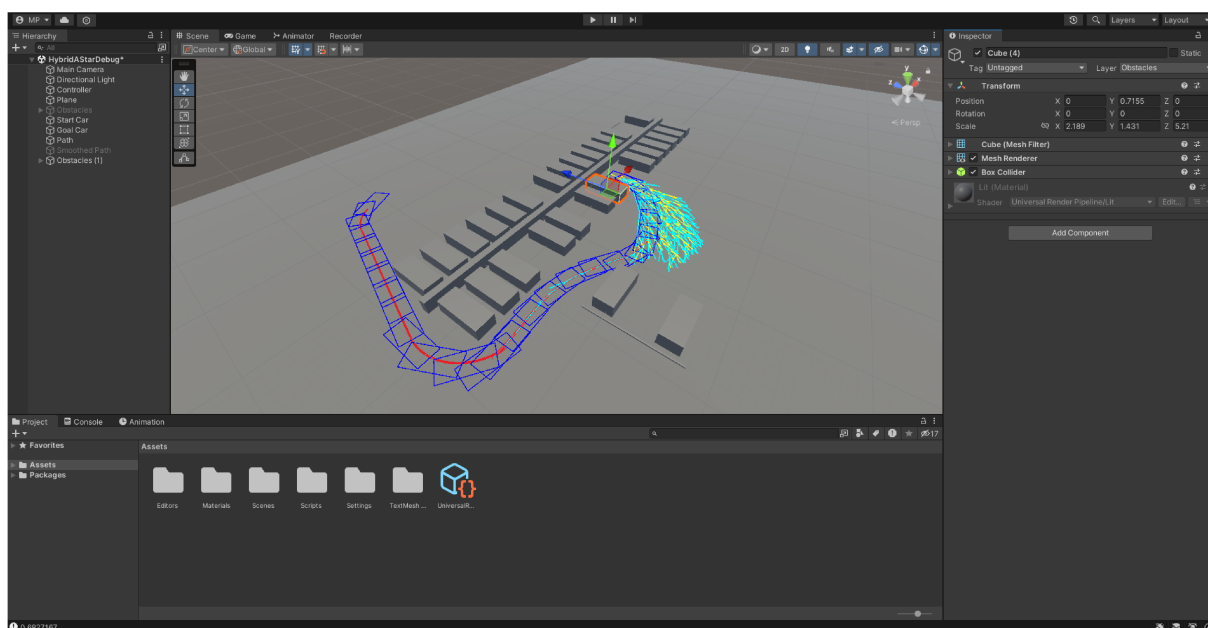
O cálculo da menor trajetória envolve a avaliação de todas as combinações possíveis, mas, com o uso de tabelas de intervalos e propriedades de continuidade dos ângulos, pode-se otimizar o processo, eliminando combinações que não correspondem à menor distância para determinados valores de ângulo e posição inicial. A métrica de Reeds-Shepp, gerada a partir dessas trajetórias, é contínua, o que permite seu uso eficiente em algoritmos de planejamento de trajetórias baseados em amostragem, tornando-se particularmente útil em robótica e veículos autônomos que operam em ambientes restritos.

## 2.6 Unity 3D como Ambiente de Simulação

O Unity 3D ([UNITY-TECHNOLOGIES, 2024](#)) é uma das plataformas mais amplamente utilizadas para o desenvolvimento de aplicações tridimensionais (3D) e bidimensionais (2D), sendo adotada tanto na indústria de jogos quanto na pesquisa acadêmica. Sua popularidade deve-se principalmente à sua interface intuitiva e às suas funcionalidades robustas, que permitem a criação de ambientes virtuais complexos e de alta fidelidade visual. No contexto deste trabalho, o Unity foi escolhido como ambiente de simulação devido à sua flexibilidade para o desenvolvimento de ambientes virtuais. A codificação no Unity é realizada utilizando o Microsoft .NET Framework e a linguagem de programação C# ([ALBAHARI, 2023](#)). Essa combinação permite a implementação de

algoritmos avançados e proporciona um controle detalhado sobre os objetos e cenários simulados. A plataforma também conta com uma ampla biblioteca de recursos e ferramentas para o gerenciamento de física, colisão e animações, tornando-a ideal para o desenvolvimento de sistemas de navegação autônoma, como veículos, que exigem precisão e realismo nas interações com o ambiente.

Figura 7 – Interface do Unity com painel de hierarquia, cena, inspetor e projeto.



Fonte: Autoria própria (2024).

A interface do Unity é organizada de forma a otimizar o fluxo de trabalho durante o desenvolvimento. Como ilustrado na Figura 7, a área principal é dividida em painéis que facilitam o acesso aos componentes e objetos presentes no projeto. No painel central, encontra-se a visualização da cena (*scene*), onde o desenvolvedor pode manipular os objetos tridimensionais, ajustar suas propriedades e visualizar em tempo real as mudanças realizadas. À esquerda, o painel de hierarquia (*Hierarchy*) lista todos os objetos atualmente presentes na cena, incluindo elementos fundamentais como câmeras, luzes e objetos físicos, essenciais para a simulação de ambientes virtuais. Na parte inferior, o painel de projeto (*Project*) contém todos os recursos importados ou criados no Unity, como *scripts*, materiais, texturas e modelos tridimensionais, organizados em uma estrutura de pastas que facilita o acesso e o gerenciamento.

O conceito de *GameObject* é central na arquitetura do Unity. Um *GameObject* representa um objeto na cena que pode ser qualquer entidade, desde um carro autônomo a uma árvore ou uma simples câmera. Esses objetos são fundamentais para a criação de cenas interativas e podem conter múltiplos componentes que definem suas características e comportamentos. Cada componente de um *GameObject* acrescenta funcionalidades específicas, como a capacidade de detectar colisões, responder a entradas do usuário ou

exibir gráficos na tela. No desenvolvimento de simulações de veículos autônomos, por exemplo, um *GameObject* pode representar o veículo em si, enquanto os componentes anexados a ele determinam suas propriedades físicas, visuais e de controle.

Uma das maiores forças do Unity está na capacidade de definir o comportamento dos *GameObjects* por meio de *scripts*, escritos predominantemente em C#. Um *script* é um arquivo de código que permite ao desenvolvedor definir as ações e reações dos objetos na cena, estabelecendo como eles interagem com o ambiente e entre si. A escolha de C# como linguagem de programação para *scripts* no Unity se deve à sua eficiência e à integração nativa com a plataforma, o que permite a execução de códigos complexos em tempo real, sem comprometer a performance. No desenvolvimento de veículos autônomos, *scripts* em C# são utilizados para implementar algoritmos de planejamento, como o Hybrid A\*, permitindo ao veículo traçar rotas e reagir a obstáculos de forma dinâmica. Por meio da orientação a objetos oferecida pela linguagem C#, o Unity facilita a organização do código em classes, possibilitando a criação de módulos reutilizáveis e a manutenção do código.

No Unity, a cena (*scene*) representa o ambiente virtual onde a simulação ocorre. É no contexto de uma cena que todos os *GameObjects* são posicionados, ajustados e configurados para interagir uns com os outros de acordo com as regras definidas pelos *scripts*. Cada cena pode representar um cenário distinto, como uma pista de teste para veículos autônomos, com diferentes configurações de obstáculos e elementos visuais que contribuem para a complexidade do ambiente. A estrutura modular das cenas permite que o desenvolvedor organize e teste várias configurações do ambiente de forma isolada, facilitando o desenvolvimento iterativo e o ajuste fino dos parâmetros da simulação.

A modularidade do Unity também se reflete na utilização de *layers*, ou camadas, que facilitam a organização e o controle de interações entre os objetos. Por exemplo, em uma simulação de navegação autônoma, é possível definir camadas específicas para o veículo, para os obstáculos e para os elementos do cenário. Dessa forma, as interações, como colisões, podem ser habilitadas ou desabilitadas com base nas camadas às quais os objetos pertencem, otimizando o cálculo de trajetórias e o desempenho geral da simulação.

Além disso, o Unity oferece ferramentas visuais, como o *LineRenderer*, que permitem a visualização direta dos dados de navegação no ambiente, incluindo trajetórias e pontos de controle em tempo real. Esse recurso é particularmente útil na validação de algoritmos de planejamento, uma vez que permite ao desenvolvedor acompanhar o desempenho e a resposta dos veículos em relação aos obstáculos e ao caminho planejado. A possibilidade de alterar as configurações do ambiente em tempo real é outra característica essencial do Unity, que proporciona ao desenvolvedor a flexibilidade de ajustar e testar diferentes cenários e configurações de maneira ágil e eficaz.

Em suma, o Unity configura-se como uma plataforma poderosa e versátil, ideal para a simulação de sistemas de navegação autônoma, permitindo a criação de ambientes

controlados e a implementação de algoritmos complexos em C#. A flexibilidade proporcionada pela estrutura modular e pela integração com C# torna o Unity uma ferramenta poderosa para o desenvolvimento de simulações precisas e de fácil controle.

## 3 Implementação do Algoritmo Hybrid A\*

Este capítulo é dedicado à apresentação detalhada da teoria por trás do algoritmo Hybrid A\*, explorando suas origens, princípios de funcionamento e aspectos técnicos fundamentais. A seção inicial aborda a evolução dos algoritmos de busca, contextualizando o desenvolvimento do Hybrid A\* como uma extensão do algoritmo A\* tradicional, projetada para lidar com restrições não-holonômicas específicas de veículos autônomos. Em seguida, o capítulo detalha as heurísticas, estratégias de expansão de nós, e outras características que tornam o Hybrid A\* adequado para a geração de trajetórias suaves e seguras em ambientes complexos.

Após a fundamentação teórica, são descritas as particularidades da implementação do Hybrid A\* desenvolvida neste trabalho. Nessa parte, são destacadas as diferenças entre a versão tradicional do algoritmo e as adaptações realizadas para atender às especificidades do ambiente de simulação e aos requisitos do projeto. Essas adaptações incluem ajustes na heurística e modificações no processo de expansão de nós para otimizar o desempenho no contexto simulado.

### 3.1 Evolução dos Algoritmos de Busca

A evolução dos algoritmos de busca ao longo das últimas décadas tem sido fundamental para o desenvolvimento de soluções de planejamento de trajetórias em veículos autônomos. A busca pelo caminho mais eficiente e seguro exige algoritmos que possam lidar não apenas com a minimização de custos de percurso, mas também com as restrições impostas pela dinâmica e pela cinemática dos veículos. Esse cenário levou ao surgimento de métodos que consideram, de forma progressiva, essas restrições, culminando na formulação do algoritmo Hybrid A\*.

Inicialmente, algoritmos clássicos de busca, como o de [Dijkstra \(1959\)](#), foram amplamente utilizados devido à sua abordagem sistemática de explorar todos os caminhos possíveis em um grafo ponderado. No entanto, esses algoritmos não consideram heurísticas para guiar a busca, o que pode resultar em um custo computacional elevado quando aplicados a problemas de grande escala.

Com o objetivo de otimizar a eficiência de busca, surgiu o algoritmo A\*, que incorpora heurísticas para priorizar a exploração de nós que aparentam estar mais próximos do objetivo ([HART; NILSSON; RAPHAEL, 1968](#)). Esse aprimoramento reduziu o espaço de busca e tornou o A\* uma das abordagens mais populares para o planejamento de trajetórias.

Ainda assim, o A\* convencional apresenta limitações quando aplicado a veículos autônomos, pois assume liberdade de movimento que não corresponde à realidade de sistemas sujeitos a restrições não-holonômicas, como carros e robôs móveis.

Para lidar com essas limitações, o Hybrid A\* foi proposto como uma extensão do A\*, adaptado para considerar as restrições de movimentação de veículos em um espaço contínuo. O Hybrid A\* mantém a estrutura de busca baseada em grafos do A\*, mas adiciona modelos cinemáticos que representam com precisão as capacidades e limitações do veículo, gerando trajetórias viáveis mesmo em ambientes com restrições de espaço e mobilidade. Essa abordagem provou ser eficaz em competições como o DARPA Urban Challenge ([MONTEMERLO et al., 2008](#)), demonstrando sua aplicabilidade em cenários de navegação autônoma complexos.

Em resumo, o desenvolvimento do Hybrid A\* representa a convergência de métodos clássicos de busca com a necessidade de adaptação para cenários de mobilidade restrita, combinando heurísticas e modelos de restrição que permitem uma navegação autônoma eficiente e segura em ambientes desafiadores.

### 3.1.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra, proposto por [Dijkstra \(1959\)](#), é um dos métodos fundamentais para a busca de caminhos de menor custo em grafos ponderados. Sua abordagem baseia-se na exploração sistemática de todos os caminhos possíveis a partir de um nó inicial, garantindo que o caminho de menor custo seja encontrado. Esse processo é essencial para problemas onde a precisão do custo total é prioridade, como na navegação em redes e sistemas de transporte. No entanto, ele apresenta limitações que incentivaram o desenvolvimento de métodos mais avançados, como o A\* e o Hybrid A\*.

O algoritmo opera atribuindo um custo inicial de zero ao nó de partida, enquanto todos os demais nós recebem um custo infinito. Em seguida, ele seleciona o nó com o menor custo acumulado para expansão, calculando o custo total para alcançar cada um dos nós vizinhos. Caso esse custo acumulado seja menor do que o valor já registrado para um nó vizinho, a atualização é realizada, e o nó atual passa a ser o predecessor do vizinho. A cada passo, o nó expandido é marcado como processado e o processo se repete até que todos os nós tenham sido explorados ou o nó de destino seja alcançado.

### 3.1.2 Heurística e Algoritmo A\*

Na ciência da computação, uma heurística é definida como uma função que estima o custo de um determinado estado até o estado objetivo, guiando a busca de maneira eficiente ao reduzir o espaço de exploração ([RUSSELL; NORVIG; DAVIS, 2010](#)). A heurística atua como um mecanismo de “atalho”, permitindo que o algoritmo direcione sua

busca para as regiões mais promissoras do espaço, minimizando a exploração de caminhos que provavelmente não levam a uma solução eficiente. Para ser eficaz, uma heurística precisa ser admissível, ou seja, nunca deve superestimar o custo real até o objetivo, garantindo que a solução encontrada seja ótima.

O algoritmo A\*, conforme descrito no Algoritmo 3, aplica a heurística combinando-a com o custo acumulado de cada nó, definindo uma função de avaliação  $f(n) = g(n) + h(n)$ , onde  $g(n)$  representa o custo real desde o nó inicial até o nó  $n$ , e  $h(n)$  é a estimativa do custo de  $n$  até o objetivo. Ao expandir preferencialmente os nós com o menor valor de  $f(n)$ , o A\* balanceia o custo já percorrido com o custo estimado, direcionando a busca de forma a alcançar uma solução ótima sem a necessidade de explorar todo o espaço de estados, como ocorre no algoritmo de Dijkstra.

Apesar de sua eficiência, o A\* também apresenta limitações quando aplicado ao planejamento de trajetórias de veículos autônomos. Embora a heurística ajude a direcionar a busca, o algoritmo A\* convencional ainda não considera as restrições cinemáticas e não-holonômicas do veículo, como o raio mínimo de curva ou a direção do movimento, limitando sua aplicabilidade a cenários onde essas restrições não são predominantes. Esse desafio motivou o desenvolvimento do Hybrid A\*, que integra as restrições do movimento diretamente no processo de busca, tornando-se uma solução mais eficaz para a navegação autônoma em ambientes complexos e restritos.

### 3.1.3 Uso de Heurísticas no Planejamento de Trajetórias

No contexto do planejamento de trajetórias, o uso de heurísticas desempenha um papel central na determinação de caminhos eficientes e viáveis entre um ponto inicial e um objetivo, especialmente em ambientes complexos e de grande escala. A heurística permite que o algoritmo de busca direcione a expansão dos nós com base em uma estimativa do custo restante até o destino, otimizando o tempo de processamento e tornando a busca mais prática para sistemas de navegação autônoma (LAVALLE, 2006).

Ao planejar trajetórias para veículos autônomos, as heurísticas mais comumente aplicadas envolvem cálculos baseados em distâncias, como a distância euclidiana ou Manhattan, que oferecem aproximações rápidas e eficazes para guiar o algoritmo de busca. Contudo, em cenários de alta complexidade, como em ambientes urbanos densamente obstruídos, essas heurísticas básicas podem ser insuficientes, pois não levam em consideração as características específicas do ambiente nem as restrições de movimentação do veículo. Nesse contexto, heurísticas mais sofisticadas são necessárias para lidar com obstáculos e com as limitações não-holônomicas dos veículos (LATOMBE, 2012).

Uma heurística bem desenhada pode evitar áreas com muitos obstáculos ou regiões de difícil navegação, priorizando caminhos que maximizam a eficiência do trajeto em

**Algoritmo 3:** Pseudocódigo para o A\*.

```

1 Discretize o ambiente e obtenha a grade com obstáculos identificados;
2 Crie um conjunto aberto (fila de prioridade) para rastrear as células a serem
  processadas;
3 Crie um conjunto fechado para registrar as células já processadas;
4 Adicione a célula inicial ao conjunto aberto;
5 enquanto conjunto aberto não estiver vazio faça
6   | Retire a célula com menor custo  $f = g + h$  do conjunto aberto;
7   | se a célula atual é o objetivo então
8   |   | Finalizar e reconstruir o caminho;
9   | fim
10  | Adicione a célula atual ao conjunto fechado;
11  | para cada vizinho da célula atual faça
12  |   | se vizinho é um obstáculo ou já estiver no conjunto fechado então
13  |   |   | Continue para o próximo vizinho;
14  |   | fim
15  |   | Calcule o custo  $g$  como o custo  $g$  da célula atual mais a distância entre a
16  |   |   | célula atual e o vizinho;
17  |   | se o vizinho está no conjunto aberto e o novo custo  $g$  é maior ou igual ao
18  |   |   | custo  $g$  do vizinho então
19  |   |   |   | Continue para o próximo vizinho;
20  |   |   | fim
21  |   | Calcule o custo heurístico  $h$  como a distância do vizinho ao objetivo;
22  |   | Atualize o custo total  $f = g + h$  do vizinho;
23  |   | Defina a célula atual como antecessor do vizinho;
24  |   | se o vizinho não está no conjunto aberto então
25  |   |   | Adicione o vizinho ao conjunto aberto;
26  |   | fim
27  | fim
28 fim

```

termos de tempo e distância. Em ambientes que incluem restrições de movimento, como o raio mínimo de curva ou as limitações de direção do veículo, as heurísticas também podem ser projetadas para incluir essas características cinemáticas, orientando o algoritmo a evitar trajetórias inviáveis. A capacidade de uma heurística de antecipar restrições ambientais e cinemáticas do veículo é um fator decisivo para a aplicabilidade do planejamento de trajetórias em sistemas de navegação autônoma.

No algoritmo A\*, a função heurística permite uma eficiência computacional elevada ao guiar o processo de expansão dos nós. No entanto, para sistemas de planejamento de trajetórias aplicados a veículos, a heurística precisa também ser capaz de lidar com cenários dinâmicos e condições variadas do ambiente. Assim, heurísticas adaptativas, que ajustam suas estimativas de custo com base nas mudanças do ambiente em tempo real, têm ganhado espaço em aplicações de navegação, proporcionando maior precisão na seleção de caminhos (CHOSSET et al., 2005). Por exemplo, heurísticas que consideram informações

sobre a presença de pedestres ou mudanças de sinalização no ambiente podem recalcular a estimativa de custo de maneira contínua, garantindo que a trajetória planejada permaneça eficiente e segura.

## 3.2 Algoritmo Hybrid A\*

Nesta seção, são apresentados os aspectos centrais do algoritmo Hybrid A\*, incluindo uma introdução teórica ao algoritmo e aos objetivos de sua aplicação no planejamento de trajetórias. Em seguida, explora-se o funcionamento do Hybrid A\*, com ênfase nas heurísticas utilizadas para guiar a busca. São discutidas as estratégias de expansão de nós, além da otimização local, que contribui para suavizar e garantir a segurança das rotas planejadas. Na subseção final, a descrição da execução expõe a implementação desenvolvida, detalhando as adaptações e melhorias específicas aplicadas para obter uma trajetória viável e otimizada no ambiente de simulação.

### 3.2.1 Introdução ao Algoritmo Hybrid A\*

O algoritmo Hybrid A\* foi desenvolvido para atender às demandas de planejamento de trajetórias em veículos autônomos que operam em ambientes complexos e não estruturados. Esse algoritmo foi projetado para solucionar um problema central: a necessidade de gerar trajetórias viáveis que considerem as restrições de movimentação do veículo, como o raio mínimo de curva e a orientação, sem sacrificar a eficiência computacional ([DOLGOV et al., 2008](#)).

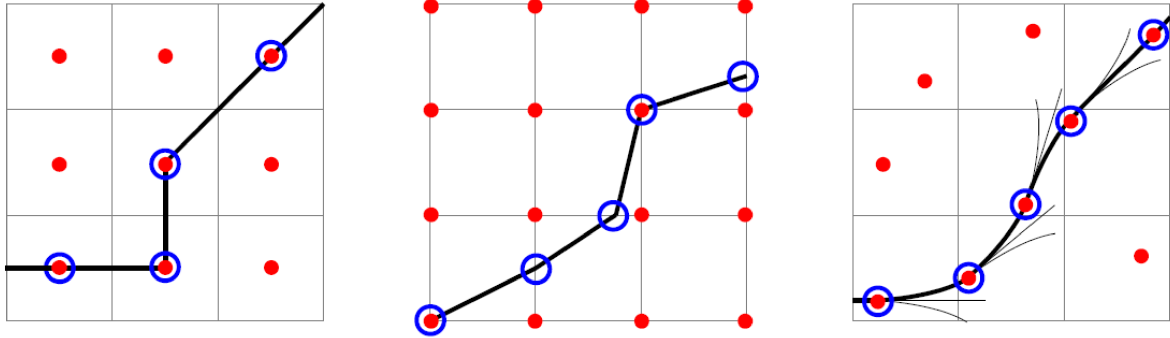
Essa abordagem foi aplicada com sucesso no DARPA Urban Challenge de 2007, onde o veículo autônomo “Junior”, desenvolvido pelo Stanford Racing Team, utilizou o Hybrid A\* para navegar com precisão e segurança em ambientes urbanos, incluindo cenários como estacionamentos e ruas bloqueadas ([MONTEMERLO et al., 2008](#)). O uso do Hybrid A\* permitiu ao veículo realizar manobras complexas, como curvas em espaços restritos e movimentos em marcha à ré, garantindo que a trajetória gerada fosse compatível com as restrições não-holônicas.

### 3.2.2 Funcionamento do Algoritmo Hybrid A\*

O Hybrid A\* diferencia-se dos métodos tradicionais ao incorporar o espaço de estados híbrido  $(x, y, \theta)$ , que representa a posição e a orientação do veículo de forma contínua, mesmo em uma grade de células discretas. Ao associar cada célula a um estado contínuo, o algoritmo consegue gerar caminhos suaves e dirigíveis, algo fundamental para veículos que devem respeitar suas próprias limitações físicas. A Figura 8 apresenta uma comparação gráfica entre o Hybrid A\* e outros algoritmos de busca, destacando como ele

associa estados contínuos a células discretas, melhorando a navegabilidade e viabilidade das trajetórias.

Figura 8 – Comparação gráfica entre algoritmos de busca: A\* tradicional, A\* com grade refinada, e Hybrid A\*.



Fonte: Dolgov et al. (2008)

Uma das principais características do Hybrid A\* é que ele não garante a solução de menor custo. Ao associar um estado contínuo a cada célula da grade, o algoritmo permite que o veículo explore movimentos precisos dentro de cada célula, mas, como resultado, perde-se a garantia de otimalidade (DOLGOV et al., 2008). Esse compromisso é necessário para que as trajetórias resultantes sejam adequadas às limitações físicas do veículo, mantendo o processo de busca eficiente.

### 3.2.2.1 Expansão de Nós no Hybrid A\*

A expansão dos nós no Hybrid A\* é um dos aspectos que diferencia esse algoritmo das abordagens convencionais de busca, como o A\*. Enquanto o A\* expande nós em uma grade discreta, assumindo que o veículo pode se mover livremente entre as células adjacentes, o Hybrid A\* adapta a expansão para considerar as restrições cinemáticas do veículo, particularmente a orientação  $\theta$ . No Hybrid A\*, cada nó representa um estado completo do veículo  $(x, y, \theta)$ , integrando tanto a posição quanto a orientação e, assim, criando trajetórias mais realistas para veículos com A cada expansão, o algoritmo gera seis novos estados possíveis a partir do nó atual, combinando três ângulos de direção com as opções de movimento para frente e para trás.

Cada um dos seis novos estados gerados em uma expansão representa uma possível configuração de  $(x, y, \theta)$ , calculada a partir de pequenos incrementos de movimento que consideram o raio de curva e a direção permitida de movimento do veículo. Esses estados são obtidos aplicando diferentes ângulos de direção, que correspondem a curvas para a esquerda, curvas para a direita e um movimento reto, considerando tanto a marcha à frente quanto a ré. O algoritmo simula esses movimentos, calculando a posição  $(x, y)$  e a orientação  $\theta$  resultantes para cada um dos seis novos estados. Esse método de expansão

proporciona uma modelagem cinemática precisa, uma vez que cada novo estado gerado representa uma posição viável que respeita as limitações de manobra do veículo.

As heurísticas utilizadas no Hybrid A\* desempenham um papel essencial ao guiar o algoritmo em direção ao objetivo de forma eficiente, evitando a exploração de estados desnecessários. No contexto do Hybrid A\*, são empregadas duas heurísticas principais: uma holonômica e outra que considera as restrições de movimento (não-holonômica).

A heurística holonômica é baseada no algoritmo A\*, mas ao contrário de uma simples distância euclidiana, calcula o custo de trajetória em um mapa de custo completo, levando em conta obstáculos no ambiente. Ela fornece ao algoritmo uma estimativa do custo total de alcançar o objetivo a partir de qualquer ponto no espaço, sem considerar as restrições de movimento do veículo, mas respeitando a viabilidade do trajeto com base na disposição dos obstáculos.

A segunda heurística, não-holonômica, refina ainda mais a busca ao incorporar as restrições de movimento do veículo, penalizando estados cuja orientação ( $\theta$ ) esteja desalinhada em relação ao objetivo. Dessa forma, o algoritmo evita explorar estados que possam estar fisicamente próximos do destino, mas com orientações que inviabilizam uma aproximação direta, contribuindo para uma trajetória mais precisa e alinhada às capacidades do veículo.

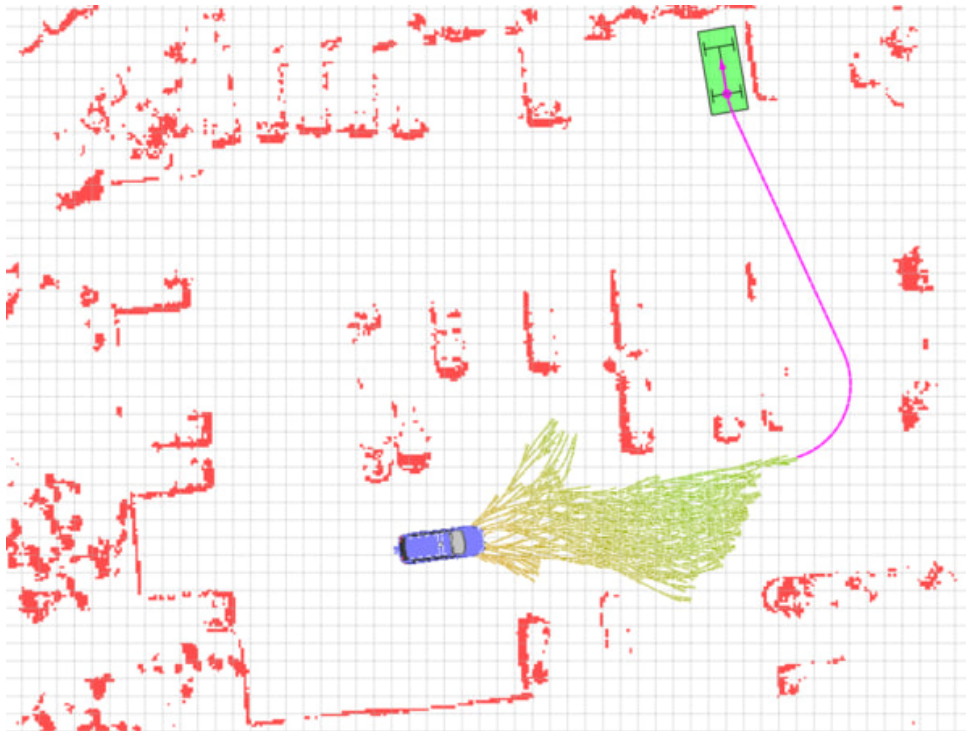
A cada expansão, o algoritmo compara os custos associados a esses novos estados para determinar quais devem ser mantidos na lista de nós abertos e quais podem ser removidos. Os nós com custos mais baixos, que incluem tanto o custo acumulado até aquele estado quanto a estimativa de custo restante até o objetivo com base na heurística, são priorizados para expansão subsequente. Além disso, são considerados os custos adicionais para manobras específicas, como ré, reversão e curvas, que aumentam o custo total em função da complexidade da trajetória e das restrições de movimento do veículo. Essas penalizações são importantes para garantir que o algoritmo priorize trajetórias diretas e minimize manobras complexas, reservando-as apenas para situações em que não há alternativas mais simples.

Quando um nó alcança uma posição suficientemente próxima do objetivo e com uma orientação compatível, o processo de expansão pode ser finalizado, e a trajetória completa é registrada como viável. Caso contrário, o algoritmo continua expandindo nós até encontrar um estado com custo aceitável que satisfaça às condições de posição e orientação necessárias, assegurando que o trajeto resultante seja tanto eficiente quanto executável pelo veículo.

### 3.2.2.2 Expansão Analítica usando as Curvas de Reeds-Shepp

Uma das extensões do Hybrid A\* é a expansão analítica baseada nas curvas de Reeds-Shepp, que permite ao algoritmo gerar uma trajetória ideal para alcançar o objetivo de forma eficiente. A cada etapa, o algoritmo tenta conectar diretamente o estado atual do veículo ao estado objetivo, utilizando a trajetória de menor comprimento possível das curvas de Reeds-Shepp. Quando a trajetória gerada é viável e livre de colisões, ela é incorporada ao percurso final, permitindo que o algoritmo complete a trajetória sem a necessidade de expansões adicionais. Esse processo otimiza a busca, reduzindo o número de cálculos necessários para finalizar o trajeto. A Figura 9 ilustra a expansão analítica com o uso das curvas de Reeds-Shepp.

Figura 9 – Exemplo de Expansão Analítica usando Reeds-Shepp.



Fonte: [Dolgov et al. \(2008\)](#)

A vantagem da expansão analítica é que ela permite alcançar o estado objetivo com maior precisão, especialmente em situações onde o algoritmo se aproxima do objetivo com um ângulo adequado. Contudo, por razões computacionais, a expansão analítica é geralmente aplicada apenas em nós próximos ao objetivo, onde a probabilidade de sucesso é maior e o ganho de eficiência se torna mais significativo.

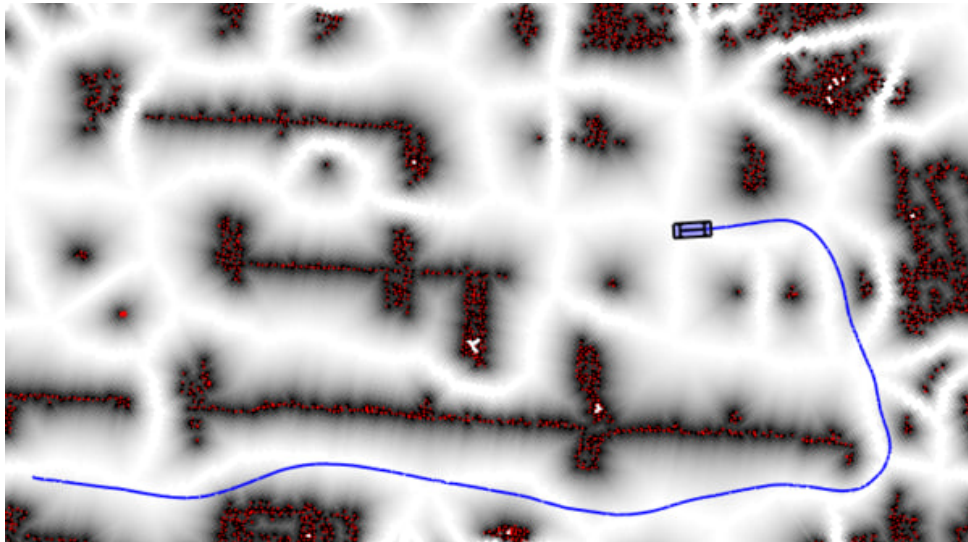
### 3.2.2.3 Campo de Voronoi

Para garantir que o veículo mantenha uma distância segura de obstáculos ao longo de sua trajetória, o Hybrid A\* incorpora um campo de potencial baseado no diagrama de

Voronoi. Esse campo, denominado Campo de Voronoi, é projetado para ajustar o custo da trajetória de acordo com a proximidade de obstáculos, promovendo uma distância segura em passagens estreitas sem impedir a navegação. O Campo de Voronoi é calculado como uma função do diagrama de Voronoi do espaço de obstáculos, o que permite ajustar o potencial do campo para que o veículo navegue com segurança em áreas complexas e de alta densidade de obstáculos, conforme a Figura 10.

A implementação do Campo de Voronoi envolve penalizar trajetórias que passam muito próximas de obstáculos, ajustando o custo das células ao longo do caminho para incentivar o veículo a se mover em áreas com maior espaço livre. Em situações onde passagens estreitas são inevitáveis, o Campo de Voronoi minimiza as áreas de alto potencial, assegurando que o caminho permaneça livre de colisões e que a navegação seja precisa.

Figura 10 – Exemplo do Campo de Voronoi aplicado em um ambiente com obstáculos.



Fonte: [Dolgov et al. \(2008\)](#)

#### 3.2.2.4 Otimização Local e Suavização

Embora o Hybrid A\* seja capaz de gerar trajetórias viáveis ao expandir nós considerando as restrições de movimento do veículo, as soluções iniciais produzidas pelo algoritmo podem não ser suficientemente suaves para garantir conforto e eficiência durante a execução da trajetória ([DOLGOV et al., 2008](#)). A trajetória gerada pela busca inicial muitas vezes possui mudanças bruscas de direção e pequenos desvios que, embora viáveis, podem impactar negativamente na estabilidade do veículo e no consumo de energia. Para resolver esse problema, o Hybrid A\* incorpora uma fase de otimização local e suavização, conhecida como *Local Optimization and Smoothing*, que visa refinar a trajetória resultante, tornando-a mais fluida e natural.

A otimização local é realizada após a geração da trajetória inicial, por meio de um processo de minimização de custo que aplica técnicas de gradiente descendente,

conforme demonstrado por [Dolgov et al. \(2008\)](#). A função de custo utilizada penaliza principalmente mudanças abruptas na direção e na curvatura, promovendo uma trajetória que respeita a continuidade e reduz o número de variações bruscas de  $\theta$ . Esse processo de suavização local ajusta iterativamente os pontos ao longo da trajetória, reduzindo desvios e removendo segmentos desnecessários, o que resulta em uma trajetória mais curta e com menor curvatura. A suavização é conduzida de forma que cada ajuste feito nos pontos intermediários da trajetória considere tanto a viabilidade cinemática quanto a ausência de colisões, mantendo assim a segurança e a aderência às restrições de mobilidade do veículo.

Além de melhorar a eficiência da trajetória, a suavização tem impacto direto no conforto e na segurança, especialmente em ambientes urbanos onde a trajetória precisa ser precisa e estável. O algoritmo ajusta incrementalmente a trajetória, eliminando movimentos angulosos e reduzindo a curvatura excessiva nas curvas, o que torna o movimento do veículo mais previsível e suave para manobras, como curvas em U ou aproximações a obstáculos.

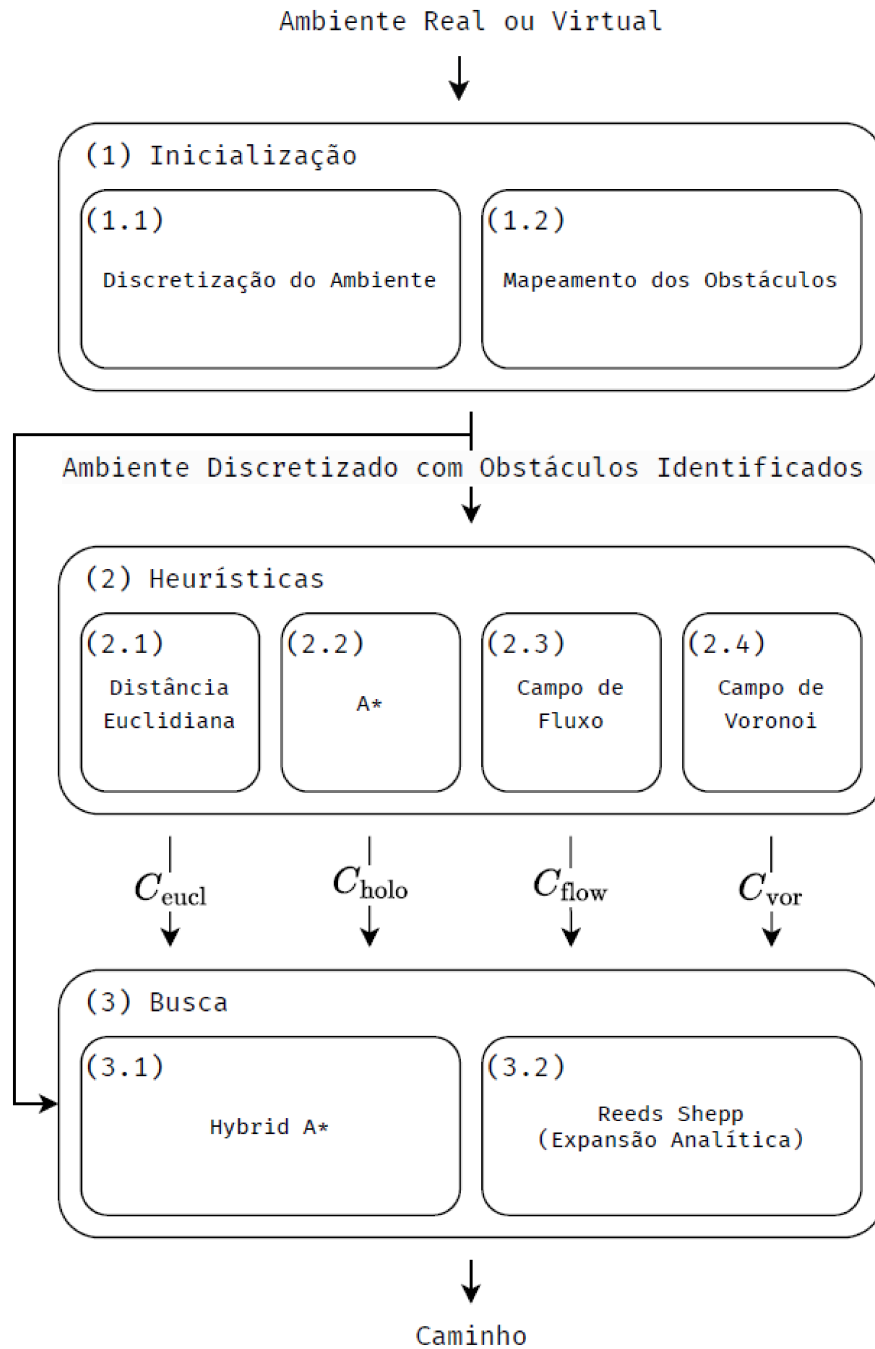
É importante destacar que essa fase de otimização e suavização adiciona uma carga computacional ao algoritmo. Cada ajuste na trajetória implica um cálculo detalhado do novo custo associado, bem como a verificação da viabilidade do movimento, considerando as limitações do espaço de busca e a presença de obstáculos. Devido a esses requisitos computacionais, a etapa de suavização pode ser ajustada para priorizar trechos críticos da trajetória, como curvas fechadas ou segmentos em áreas com alta densidade de obstáculos.

### 3.2.3 Descrição da Execução do Hybrid A\*

Esta seção descreve os passos de implementação do algoritmo Hybrid A\* utilizados neste trabalho, adaptado a partir da abordagem proposta por [Dolgov et al. \(2008\)](#) e, também, da implementação bem sucedida realizada por [Kurzer \(2016\)](#). O diagrama apresentado na Figura 11 organiza o processo de planejamento de trajetória em três fases essenciais: (1) Inicialização; (2) Cálculo das Heurísticas; e (3) Busca.

A primeira fase (Figura 11), denominada Inicialização, está representada no diagrama pelas etapas (1), (1.1) e (1.2). Nessa etapa, o ambiente de simulação é estruturado como uma grade ocupacional, onde cada célula indica uma área do espaço classificada como navegável ou ocupada por um obstáculo. Esse processo de modelagem do espaço é essencial para identificar tanto o ambiente quanto os obstáculos nele presentes. Em uma aplicação prática com sensores, as informações sobre os obstáculos seriam coletadas em tempo real, de acordo com as leituras dos sensores embarcados. Entretanto, neste estudo, o ambiente virtual da Unity disponibiliza previamente as informações de posição e dimensão dos obstáculos, permitindo uma organização mais eficiente e estável do espaço simulado. Esse espaço, uma vez discretizado em uma grade regular, serve como a estrutura base para o cálculo das heurísticas e para a execução da busca do algoritmo Hybrid A\*, assegurando que a representação do ambiente seja precisa e consistente ao longo do

Figura 11 – Diagrama da implementação do algoritmo Hybrid A\*.



Fonte: Autoria própria (2024).

processo de planejamento de trajetória.

A etapa subsequente é o cálculo das heurísticas, representado na etapa (2) do diagrama apresentado na Figura 11, que abrange as heurísticas indicadas nas etapas (2.1) a (2.4). Embora na versão proposta por Kurzer (2016) não haja uma fórmula específica para combinar as heurísticas, permitindo ajustes conforme o contexto, nesta implementação foi desenvolvida uma função heurística prática, definida empiricamente pela Equação 3.1, que se mostrou eficaz nos testes conduzidos. Cada heurística é calculada com base na

grade ocupacional e desempenha um papel específico na orientação da busca, contribuindo com informações sobre a proximidade do destino e a presença de obstáculos, de forma a otimizar o trajeto. A função heurística combinada é formulada como:

$$h(x, y) = \max(C_{\text{eucl}}(x, y), C_{\text{holo}}(x, y)) + C_{\text{flow}}(x, y) \cdot C_{\text{vor}}(x, y) \cdot 0,5 + C_{\text{flow}}(x, y) \cdot 1,1 \quad (3.1)$$

onde:

- $C_{\text{eucl}}(x, y)$ : Distância Euclidiana, que fornece uma estimativa direta da proximidade do veículo ao destino, calculando a distância em linha reta entre cada posição da célula e a posição final do veículo.
- $C_{\text{holo}}(x, y)$ : Heurística Holonômica, calculada com o algoritmo A\* invertido, conforme o Algoritmo 3. Aqui, o destino final do veículo torna-se o ponto de partida, e a posição inicial do veículo é o objetivo. A execução prossegue até que todas as células tenham um custo de movimentação, formando um mapa de custo holonômico.
- $C_{\text{flow}}(x, y)$ : Campo de Fluxo, gerado pelo Algoritmo 1. Calculado sobre a grade ocupacional, o campo expande-se a partir da posição final do veículo, formando um gradiente de valores que sugere uma direção preferencial de movimento em cada célula.
- $C_{\text{vor}}(x, y)$ : Campo de Voronoi, calculado pelo Algoritmo 2, que penaliza trajetórias próximas aos obstáculos. Quanto mais próximo um ponto está de um obstáculo, maior será o valor de penalidade, incentivando o veículo a seguir por áreas mais seguras.

Diferentemente do algoritmo descrito nas seções anteriores, esta versão omite a heurística não holonômica, cuja inclusão, segundo resultados práticos e os próprios dados obtidos por Kurzer (2016), teria um impacto mínimo na eficácia geral. Em contrapartida, foi adicionada a heurística baseada no campo de fluxo, que se mostrou eficaz para guiar o veículo ao longo de trajetórias otimizadas, especialmente em ambientes com obstáculos complexos.

A fase de busca, representada na etapa (3) do diagrama, é executada pelo algoritmo Hybrid A\*, ilustrada pela etapa (3.1) e detalhada no Algoritmo 4. Nessa etapa, uma fila de prioridade gerencia os nós abertos, priorizando aqueles de menor custo total  $f$ , calculado com base na função heurística  $h$ . Cada nó é expandido conforme essa prioridade, com o custo total atualizado e novos estados avaliados a cada iteração.

Além disso, uma lista fechada é utilizada para gerenciar os estados  $(x, y, \theta)$  já processados, onde  $x$  e  $y$  são considerados em relação à célula na grade, e o ângulo  $\theta$  é discretizado, sendo arredondado para otimizar o gerenciamento de estados. Essa lista evita

reprocessamento de estados redundantes, garantindo eficiência ao manter o controle dos nós já visitados.

Para assegurar que as trajetórias respeitem as limitações físicas do veículo, o modelo cinemático descrito na Seção 2.1 é aplicado na expansão dos nós, incorporando o ângulo de direção e o raio mínimo de curva. A busca continua até que o nó correspondente ao destino seja alcançado (dada as aproximações de posição e ângulo), momento em que a trajetória é registrada e a busca finalizada.

Durante a busca, a expansão analítica com curvas de Reeds-Shepp, representada pela etapa (3.2) no diagrama, é utilizada para calcular a trajetória de menor custo entre o nó atual e a posição final. Esta expansão é realizada sobre o espaço contínuo e permite manobras complexas, como curvas fechadas e deslocamentos em marcha à ré, essenciais em ambientes restritos. A aplicação de Reeds-Shepp ocorre geralmente em nós próximos ao destino, onde uma conexão direta ao objetivo pode ser calculada, economizando nós adicionais e refinando o trajeto final. Embora seja intensiva computacionalmente, a expansão de Reeds-Shepp foi aplicada a cada nó expandido nesta implementação para priorizar a precisão e segurança do trajeto, considerando que a eficiência temporal não é o foco deste trabalho.

Com esses passos, a trajetória final é gerada e pode ser ajustada, incluindo a aplicação de suavização, como descrito na Subseção 3.2.2.4. No entanto, optou-se por omitir a etapa de suavização neste trabalho, pois os resultados obtidos já atenderam aos objetivos propostos de maneira satisfatória.

**Algoritmo 4:** Pseudocódigo para o Hybrid A\*.

```

1 Discretize o ambiente e obtenha a grade com obstáculos identificados;
2 Crie um conjunto aberto (fila de prioridade) para rastrear os estados a serem
  processados;
3 Crie um conjunto fechado para registrar os estados  $(x, y, \theta)$  já processados;
4 Crie o estado inicial com a posição e orientação do veículo;
5 Adicione o estado inicial ao conjunto aberto;
6 enquanto conjunto aberto não estiver vazio faça
7   | Retire o estado com menor custo  $f = g + h$  do conjunto aberto;
8   | Execute o caminho de Reeds-Shepp entre o estado atual e o objetivo;
9   | se o caminho de Reeds-Shepp não estiver obstruído então
10  |   | Finalize e retorne o caminho;
11  | fim
12  | se o estado atual está próximo o suficiente do objetivo (posição e ângulo
13  |   arredondados) então
14  |   | Finalize e retorne o caminho;
15  |   | fim
16  | se o estado atual já estiver no conjunto fechado então
17  |   | Continue;
18  |   | fim
19  | para cada sucessor gerado (6 novos estados) do estado atual faça
20  |   | Calcule os custos  $g$  e  $h$  do sucessor;
21  |   | se sucessor  $(x, y, \theta)$  já estiver no conjunto fechado então
22  |   |   | Continue;
23  |   |   | fim
24  |   | se o custo  $g$  do sucessor é maior do que o menor custo  $g$  para a mesma
25  |   |   posição  $(x, y, \theta)$  então
26  |   |   | Continue;
27  |   |   | fim
28  |   | Adicione o sucessor ao conjunto aberto para processamento;
29 fim

```

## 4 Ambiente Virtual e Integração do Algoritmo Hybrid A\*

Este capítulo descreve a implementação e a integração do algoritmo *Hybrid A\** no ambiente virtual Unity, utilizando a linguagem de programação C#. O objetivo é demonstrar como o Unity, com suas funcionalidades de simulação física e renderização tridimensional, permite a criação de um ambiente para a simulação de veículos autônomos. A implementação foi realizada com foco na modularidade e na eficiência, utilizando componentes e *scripts* personalizados para garantir a flexibilidade e a escalabilidade do projeto. Nesta capítulo, cada parte essencial do sistema será detalhada, incluindo a configuração do ambiente no Unity, a definição das classes e objetos utilizados, e as técnicas empregadas para o cálculo das trajetórias e o desvio de obstáculos.

Além disso, será apresentada uma análise detalhada das implementações realizadas no contexto do algoritmo Hybrid A\* aplicado ao planejamento de trajetórias em veículos autônomos. Para tanto, as classes desenvolvidas para essa implementação serão descritas individualmente, acompanhadas dos resultados específicos observados em cada componente. A análise é estruturada de modo a examinar separadamente o desempenho de cada classe fundamental do sistema, possibilitando uma visão aprofundada das contribuições individuais de cada módulo. A partir dessa abordagem modular, é possível avaliar o impacto e a eficiência de cada parte na construção da solução final, que será posteriormente visualizada e discutida com base nos resultados integrados de todas as etapas.

### 4.1 Dimensões do Veículo e Cálculo de Movimento

As dimensões do veículo simulado são baseadas no Tesla Model S, conforme apresentado na Tabela 1. Esse modelo foi escolhido pela sua relevância no contexto de veículos autônomos e elétricos, fornecendo uma base realista para simular trajetórias e manobras em um ambiente virtual. A Figura 12 ilustra as principais medidas do veículo, incluindo comprimento, largura, altura, entre-eixos e balanços dianteiro e traseiro.

Figura 12 – Dimensões do Tesla Model S.

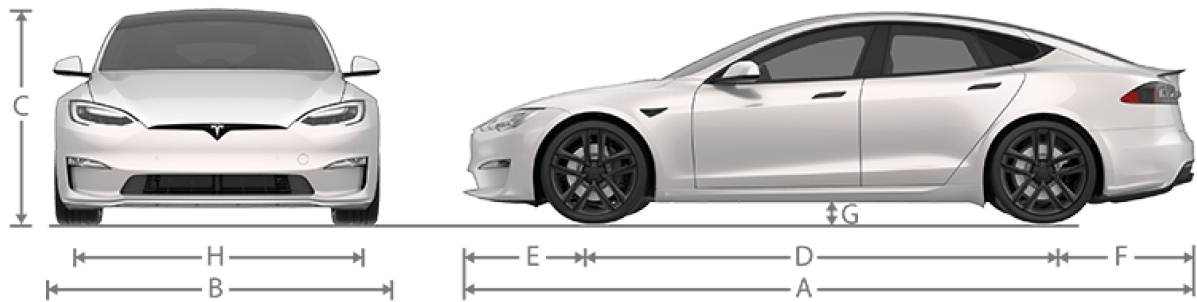
Fonte: [Tesla \(2021\)](#).

Tabela 1 – Dimensões do Tesla Model S

| Identificação | Descrição                                    | Medida (m) |
|---------------|----------------------------------------------|------------|
| A             | Comprimento Total                            | 5,021      |
| B             | Largura Total (incluindo retrovisores)       | 2,189      |
|               | Largura Total (excluindo retrovisores)       | 1,987      |
| C             | Altura Total (suspensão média)               | 1,431      |
| D             | Entre-eixos                                  | 2,960      |
| E             | Balanço Dianteiro                            | 961        |
| F             | Balanço Traseiro                             | 1,100      |
| H             | Distância entre Rodas (dianteira e traseira) | 1,690      |
| G             | Distância ao Solo                            | 0,126      |

Fonte: [Tesla \(2021\)](#)

Para os cálculos de movimentação, utiliza-se a classe `CarModel` (Anexo D), que encapsula a lógica de movimento descrita nas Equações 2.6 a 2.10. Essa classe permite estimar a nova posição e orientação do veículo com base em uma posição inicial e parâmetros de movimento específicos. Essa classe utiliza a classe `Movement` (Anexo C) para obter as informações de cada movimento, incluindo a distância (`Distance`), o tipo de direção (`Steering`) — `Left`, `Straight` ou `Right` — e o tipo de marcha (`Gear`), que pode ser `Forward` ou `Backward`.

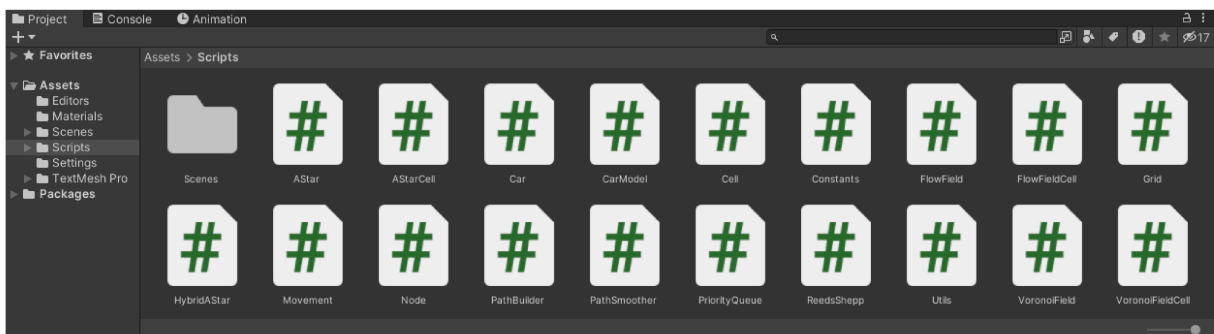
Cada segmento de trajetória do Reeds-Shepp e cada estado do Hybrid A\* são representados por instâncias da classe `Movement`, permitindo que a classe `CarModel` calcule a posição futura do veículo ao longo de cada movimento. A classe `PathBuilder` (Anexo O) também utiliza a classe `CarModel` para gerar uma lista de pontos intermediários entre os a partir de uma lista de movimentos, retornando a posição e o ângulo do veículo em cada ponto da trajetória. Isso é especialmente útil para o cálculo da trajetória do Reeds-Shepp, que consiste em até cinco movimentos, pois permite obter uma representação detalhada da posição e orientação do veículo ao longo de toda a trajetória.

Essa estrutura modular traz diversas vantagens: facilita a integração e adaptação dos movimentos ao Hybrid A\* e às curvas de Reeds-Shepp, permite o cálculo preciso de posições intermediárias e proporciona flexibilidade para ajustes e testes. A separação das funcionalidades em classes como `CarModel`, `Movement` e `PathBuilder` também melhora a organização do código, simplificando futuras expansões e adaptações para diferentes tipos de veículos e algoritmos de planejamento.

## 4.2 Divisão dos *Scripts* e Cenários de Simulação

Para organizar de forma eficiente o desenvolvimento e a integração dos componentes do algoritmo Hybrid A\* no ambiente Unity, cada *script* foi atribuído a uma função específica e armazenado em pastas de acordo com sua funcionalidade. A Figura 13 apresenta a lista de *scripts* desenvolvidos, excluindo aqueles diretamente anexados a objetos para acessar recursos do Unity por meio da classe `MonoBehaviour`. Esses últimos encontram-se na pasta “Scenes”, pois são *scripts* responsáveis pela comunicação com o Unity e pela obtenção de dados da cena.

Figura 13 – *Scripts* utilizados para a implementação do Hybrid A\*.



Fonte: Autoria própria (2024).

A estrutura do projeto foi planejada para que cada *script* se concentrasse em uma tarefa específica, seguindo o fluxo descrito no diagrama da Figura 11. Cada etapa foi implementada em *scripts* individuais, conforme listado na Tabela 2, que identifica cada bloco do diagrama e seus respectivos *scripts*. Essa abordagem promove modularidade e facilita a manutenção do código. Por exemplo, alguns *scripts* são dedicados exclusivamente à lógica da grade para a discretização do espaço, enquanto outros implementam algoritmos específicos, como A\*, Reeds-Shepp, Brushfire (Campo de Fluxo) e o Campo de Voronoi.

Tabela 2 – Organização dos *scripts* de acordo com as etapas de implementação do Hybrid A\*.

| <b>Etapas</b> | <b><i>Scripts</i></b>          |
|---------------|--------------------------------|
| (1)           | Grid; Cell                     |
| (2.2)         | AStar; AStarCell               |
| (2.3)         | FlowField; FlowFieldCell       |
| (2.4)         | VoronoiField; VoronoiFieldCell |
| (3.1)         | CarModel; HybridAStar; Node    |
| (3.2)         | ReedsShepp                     |

Fonte: Autoria própria (2024).

Além disso, foram criadas cenas individuais para testar cada um desses componentes de maneira isolada. A divisão das cenas foi feita para contemplar funcionalidades específicas, incluindo:

- criação da grade (discretização) e detecção de obstáculos;
- implementação das curvas de Reeds-Shepp para movimentação;
- geração do Campo de Fluxo para orientação de trajetórias;
- implementação do algoritmo A\* para planejamento de caminhos;
- construção do Campo de Voronoi para definição de áreas seguras de navegação;
- integração final do algoritmo Hybrid A\*.

Cada uma dessas cenas contém um *GameObject* controlador, chamado *controller*, ao qual está associado um *script* que utiliza a classe `MonoBehaviour` para obter dados diretamente do Unity. Esse controlador serve como intermediário entre a cena e os demais *scripts*, sendo responsável por executar outras classes e repassar a elas informações essenciais obtidas do Unity, como a *Layer Mask* dos obstáculos, as dimensões e a posição do ambiente a ser discretizado, entre outros. Essa estrutura modular e bem segmentada facilita o teste individual dos algoritmos e permite ajustes rápidos durante o desenvolvimento.

### 4.3 Discretização e Detecção de Obstáculos

A discretização do espaço e a detecção de obstáculos são etapas fundamentais para a implementação dos algoritmos Brushfire (Campo de Fluxo), Campo de Voronoi, A\* e, finalmente, o Hybrid A\*. O primeiro passo para esse processo é a criação de um plano discreto, denominado grade, que transforma o espaço contínuo do Unity em um espaço discreto. Para alcançar esse objetivo, foram desenvolvidas duas classes principais: **Grid**

(Anexo A) e **Cell** (Anexo B). A classe **Grid** representa o espaço global da simulação, enquanto a classe **Cell** representa cada unidade ou célula desse espaço.

Essas classes foram projetadas para oferecer flexibilidade e modularidade, permitindo que diferentes algoritmos operem sobre a grade de maneira adaptativa. Como essa estrutura é utilizada por múltiplos algoritmos que operam sobre o plano discreto, foi adotada a solução de classes genéricas em C# (ALBAHARI, 2023). Essa abordagem permite que cada algoritmo utilize uma especialização da classe **Cell**, garantindo que cada célula tenha atributos específicos conforme necessário.

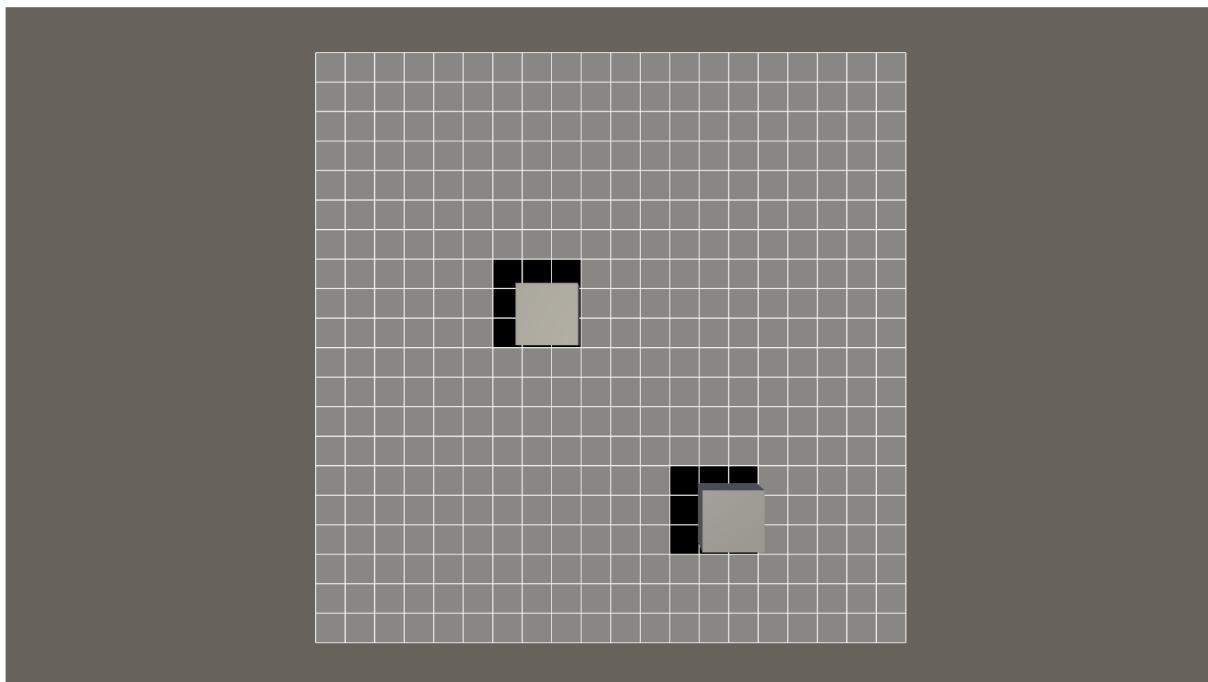
Em C#, classes genéricas são definidas para aceitar parâmetros de tipo, permitindo criar estruturas reutilizáveis com diferentes tipos de dados. A definição da classe **Grid** segue o formato `Grid<T> where T : Cell`, que indica que o tipo **T** passado como parâmetro deve herdar da classe **Cell**. Isso permite desenvolver especializações como **FlowFieldCell**, **VoronoiFieldCell** e **AStarCell**, que herdam as propriedades da classe **Cell**. A classe **Cell**, por sua vez, contém atributos fundamentais compartilhados entre todas as subclasses, como a posição e o estado de ocupação (livre ou obstruída).

Para criar a grade, são necessários o tamanho da área, o centro da grade (relativo à posição global do ambiente), o tamanho de cada célula e uma *Layer Mask* para identificar obstáculos — todos fornecidos pelo *controller*. Esses parâmetros permitem definir a dimensão e a posição exata da grade no espaço, garantindo que os obstáculos sejam detectados corretamente.

As *Layer Masks*, descritas na Seção 2.6, permitem controlar as interações entre diferentes tipos de objetos no Unity. Neste caso, a *Layer Mask* é usada para identificar e configurar obstáculos dentro do ambiente de simulação.

Durante o processo de discretização do ambiente (ou seja, a criação da grade), é verificado automaticamente se cada célula está sob a influência de um obstáculo. Para isso, utiliza-se o método **CheckCollision**, que verifica a presença de obstáculos a partir do centro de cada célula, fazendo uso da função **CheckBox** da biblioteca **Physics** do Unity. A **Physics** oferece uma série de métodos para detectar colisões e gerenciar interações físicas no ambiente 3D; no caso do **Physics.CheckBox**, essa função permite verificar se um volume definido intercepta um objeto em uma camada específica. Ao aplicar essa função no centro de cada célula e ao especificar a *Layer Mask* dos obstáculos para o **CheckBox**, é possível determinar se cada célula está obstruída.

Figura 14 – Formação da grade com células de 0,5 m, mostrando dois blocos como obstáculos e as células bloqueadas em preto.



Fonte: Autoria própria (2024).

A Figura 14 ilustra a discretização de um plano de 10x10 m, onde blocos são considerados obstáculos, identificados pela *Layer Mask* correspondente. Cada célula da grade possui 0,5 m de lado. A grade foi gerada de forma genérica com a classe `Cell`, utilizando uma estrutura `Grid<Cell>` que não é específica de nenhum algoritmo. As linhas da grade, em branco, destacam as células discretizadas, enquanto as células que representam obstáculos estão preenchidas em preto, indicando obstrução.

Por fim, a grade implementada de forma genérica permite a criação de um plano discreto que serve de base para cada um dos algoritmos, possibilitando o teste individual de cada componente antes da integração final com o Hybrid A\*. A utilização do Unity facilitou significativamente a identificação do ambiente e a detecção de obstáculos, substituindo a complexidade que, em uma situação real, exigiria o uso de sensores físicos. Esses sensores, além de serem menos precisos, estariam sujeitos a ruídos e limitações de alcance, enquanto o Unity fornece uma representação exata do ambiente, assegurando uma simulação precisa e controlada para o desenvolvimento e a validação dos algoritmos.

## 4.4 Heurísticas Holonômicas

Nesta seção, serão apresentadas as implementações das heurísticas holonômicas desenvolvidas no Unity, com o objetivo de orientar o planejamento de trajetórias do algoritmo Hybrid A\*. Cada heurística foi implementada e testada de forma independente,

permitindo uma análise detalhada de seus resultados e comportamentos.

A implementação do Campo de Fluxo visa criar um gradiente de valores que direciona a navegação em direção a uma célula-alvo específica. Já a função de custo total  $f(x, y)$  da célula, usada no algoritmo A\*, complementa o Campo de Fluxo ao minimizar problemas de aprisionamento e oferecer uma trajetória mais eficiente. Nesta seção, os resultados de cada heurística são discutidos, com observações sobre suas contribuições e limitações na navegação em diferentes cenários de teste.

#### 4.4.1 Campo de Fluxo

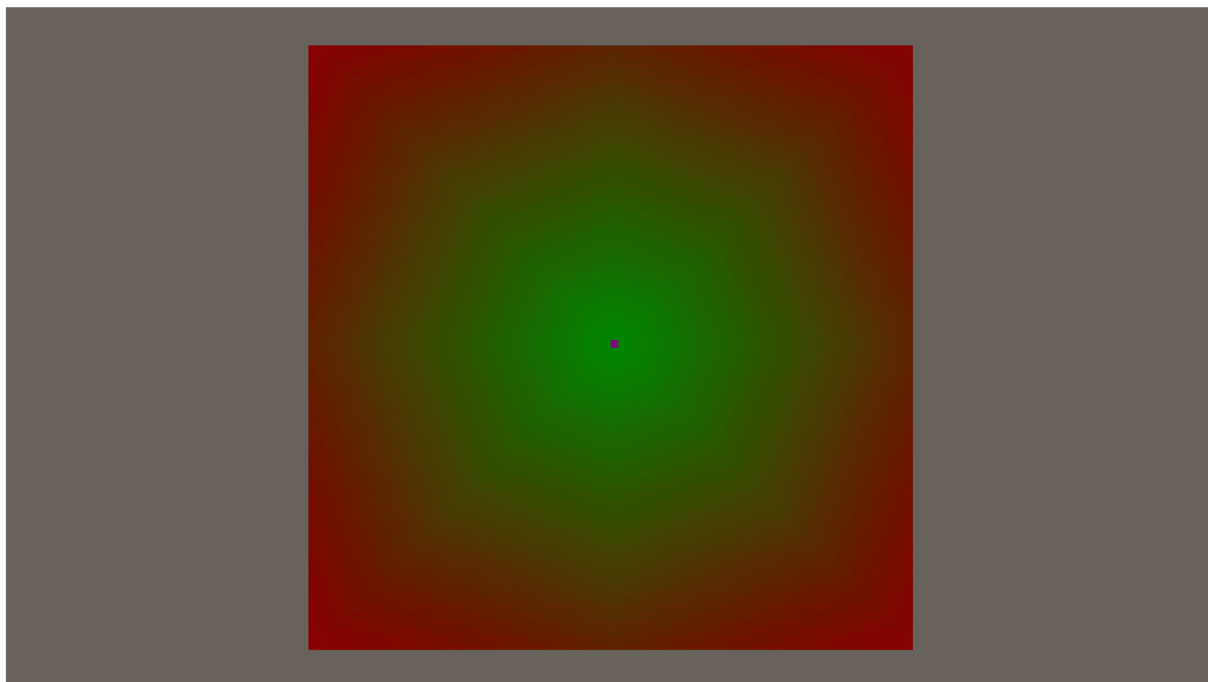
O Campo de Fluxo implementado neste trabalho foi desenvolvido para criar um gradiente de valores que direciona qualquer posição inicial em direção a uma célula-alvo específica, representando o objetivo. O algoritmo que gera o Campo de Fluxo usada está descrito no Algoritmo 1. Para esta implementação, a classe `FlowField` (Anexo I) encapsula a lógica do cálculo de propagação de valores, enquanto a classe `FlowFieldCell` (Anexo H) define cada célula individualmente, permitindo a atribuição de um custo e de vizinhos a cada célula na grade.

A grade do Campo de Fluxo foi criada de forma semelhante à descrita na Seção 4.3, com a diferença de que cada célula é representada por uma instância de `FlowFieldCell` (`Grid<FlowFieldCell>`), que armazena informações específicas para o cálculo do fluxo. A `FlowFieldCell` armazena o custo de cada célula, sua região e as células vizinhas. A célula-alvo é marcada com valor nulo, que aumenta gradualmente à medida que a distância em relação ao objetivo se expande. As células intransitáveis são designadas com um valor alto, representado por  $+\infty$ , indicando áreas que o veículo deve evitar.

Visualmente, o Campo de Fluxo oferece uma representação intuitiva do caminho até o objetivo, com a célula-alvo indicada em roxo e as demais células variando de verde a vermelho, onde o verde representa valores de custo mais baixos e o vermelho indica valores mais altos. Células intransitáveis, destacadas em azul, possuem um valor de  $+\infty$  e são desconsideradas no cálculo da trajetória.

A Figura 15 ilustra o Campo de Fluxo em um mapa menor, facilitando a observação da distribuição dos valores de custo. A célula-alvo, em roxo, atrai as demais células, orientando o fluxo em direção ao objetivo, independentemente da posição inicial.

Figura 15 – Campo de Fluxo de atração com célula objetivo no centro em roxo, variando de verde (baixo custo) a vermelho (alto custo).

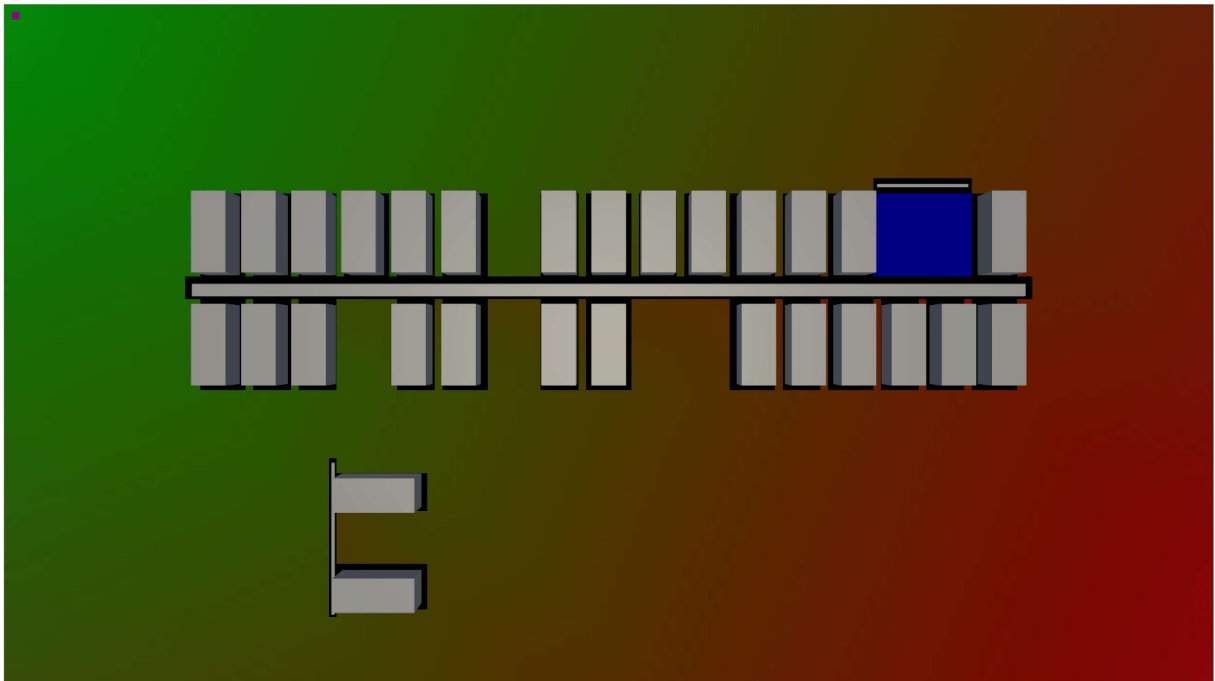


Fonte: Autoria própria (2024).

Na Figura 16, o Campo de Fluxo é aplicado em um mapa maior, com uma região de obstáculos. A presença de obstáculos cria áreas intransitáveis em azul (valor  $+\infty$ ), que o algoritmo desconsidera ao calcular o trajeto. A configuração com um obstáculo em formato de contorno em “C” destaca uma limitação do Campo de Fluxo: o veículo pode ficar “preso” dentro da estrutura, sem rota direta para o objetivo. Em situações como essa, algoritmos de busca como o A\* são mais eficazes, pois conseguem explorar rotas alternativas e contornar áreas de aprisionamento.

Ao incorporar o valor do Campo de Fluxo como uma heurística, o algoritmo Hybrid A\* é orientado a preferir caminhos com valores de fluxo mais baixos, que indicam proximidade ao objetivo. Essa heurística adicional contribui para uma navegação mais direcionada e eficiente em direção ao destino final, oferecendo uma orientação de baixo custo ao longo da trajetória. No entanto, embora o Campo de Fluxo melhore a eficiência na escolha das rotas, ele não previne diretamente colisões com obstáculos. Em cenários complexos, o Campo de Fluxo atua principalmente como uma heurística de suporte, complementando a avaliação do A\* para guiar o movimento em áreas abertas e aproximar o veículo do objetivo de maneira otimizada.

Figura 16 – Campo de Fluxo de atração com obstáculos, destacando áreas intransitáveis em azul.

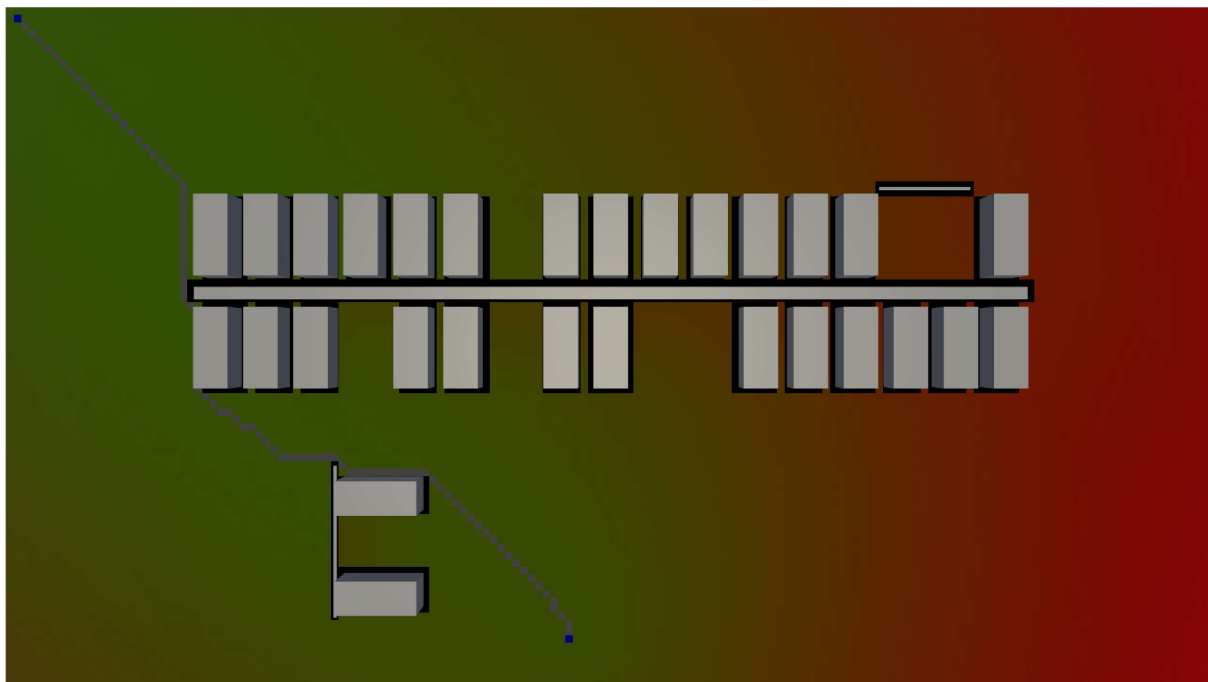


Fonte: Autoria própria (2024).

#### 4.4.2 A\*

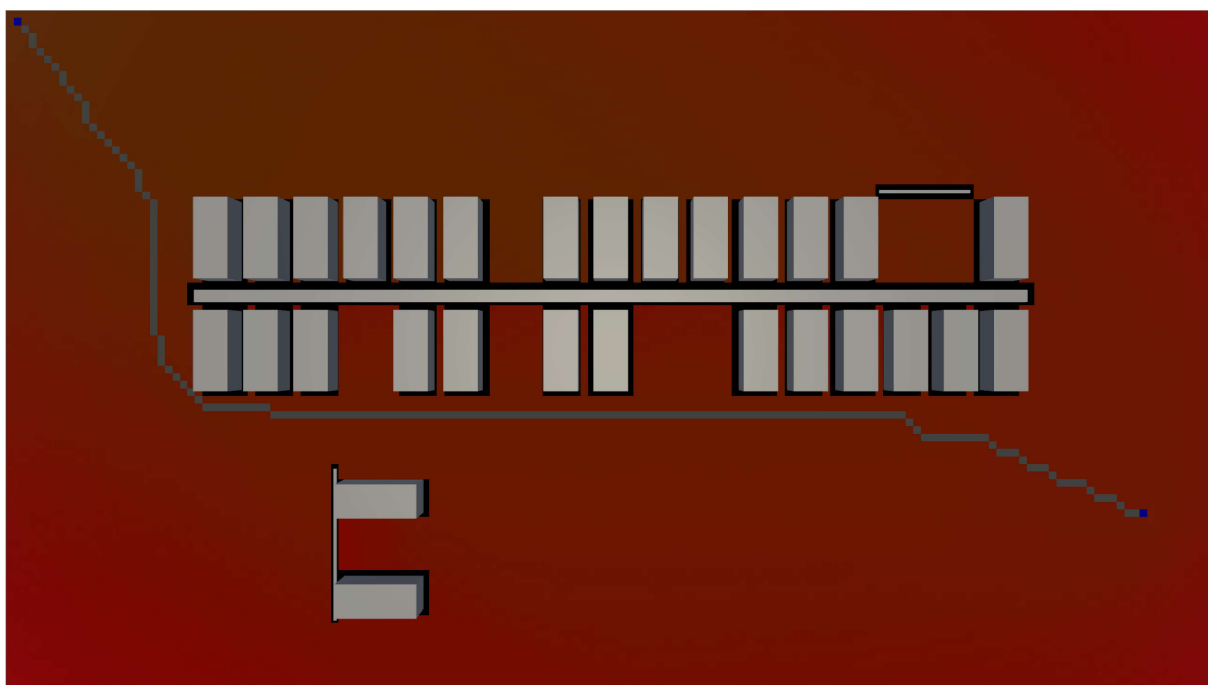
A Figura 17 e a Figura 18 apresentam a aplicação do algoritmo A\* no mesmo mapa utilizado anteriormente para o Campo de Fluxo, incluindo o formato em “C” e outros obstáculos adicionais. Em ambas as figuras, as células de início e objetivo são destacadas em azul, com o caminho encontrado pelo A\* representado em cinza, conectando a posição inicial ao destino. Na Figura 17, onde a célula inicial está mais próxima do objetivo e do contorno em “C”, a transição de cores é mais gradual, indicando uma distribuição de custos mais uniforme. Já na Figura 18, com a célula inicial posicionada mais distante do objetivo, a transição de cores não é tão gradual, refletindo um aumento mais abrupto no custo total à medida que o algoritmo avança em direção ao destino.

Figura 17 – Mapa com aplicação do A\*, célula inicial próxima ao objetivo e ao contorno em "C".



Fonte: Autoria própria (2024).

Figura 18 – Mapa com aplicação do A\*, célula inicial distante do objetivo.



Fonte: Autoria própria (2024).

O custo total de cada célula,  $f(x, y)$ , é visualizado no mapa com uma gradação de cores, variando de verde a vermelho. Valores mais baixos de  $f(x, y)$  são indicados em verde, enquanto valores mais altos aparecem em vermelho. Diferentemente do Campo de

Fluxo, a transição de cores no  $f(x, y)$  do A\* reflete a natureza heurística do algoritmo, que otimiza o caminho considerando tanto o custo acumulado ( $g(x, y)$ ) quanto a estimativa da distância ao objetivo ( $h(x, y)$ ). Essa diferença é especialmente evidente na Figura 18, onde o início distante resulta em uma transição quase imperceptível entre as cores à medida que o A\* aproxima-se do destino.

Para implementar essa lógica, foi utilizado o Algoritmo 3, cuja estrutura está encapsulada na classe **AStar** (Anexo G). A grade foi configurada de forma semelhante à descrita na Seção 4.3, mas utilizando a célula **AStarCell** (Anexo F) para armazenar as informações específicas de cada nó na grade (**Grid<AStarCell>**). A **AStarCell** guarda o custo total  $f(x, y)$ , além de dados sobre o custo acumulado e a heurística, necessários para calcular a melhor rota até o destino.

Para gerenciar os nós a serem processados, a classe **PriorityQueue** (Anexo E) é utilizada. Essa fila de prioridade organiza os nós com base em seu custo total  $f = g + h$ , onde  $g$  é o custo acumulado do caminho até o nó e  $h$  é o custo heurístico estimado até o objetivo. A **PriorityQueue** garante que os nós com menor custo total sejam expandidos primeiro.

No contexto do Hybrid A\*, o valor  $f(x, y)$  foi utilizado como uma heurística complementar para direcionar o veículo em direção ao objetivo, ajudando a evitar problemas de aprisionamento, como os observados na área em “C” no Campo de Fluxo. Contudo, devido à variação menos sutil entre células com diferentes valores de  $f(x, y)$ , o uso direto dessa heurística pode resultar em inconsistências na navegação, com o veículo possivelmente realizando desvios desnecessários.

Para mitigar esse efeito, uma alternativa considerada, mas não implementada, seria suavizar os valores de  $f(x, y)$  ao redor do caminho encontrado, reduzindo a influência das células adjacentes e criando um gradiente mais uniforme.

## 4.5 Campo de Voronoi

Na implementação do Campo de Voronoi, os resultados obtidos demonstram uma clara distinção entre as regiões de influência de cada obstáculo e as transições graduais de valores de campo ( $\rho_V$ ) conforme a distância aos obstáculos. A Figura 19 ilustra as regiões de cada obstáculo, cada uma destacada com uma cor diferente, onde cada bloco representa um obstáculo específico. Observa-se que as bordas do ambiente foram tratadas como obstáculos, formando suas próprias regiões de influência.

Figura 19 – Regiões de influência dos obstáculos no Campo de Voronoi.



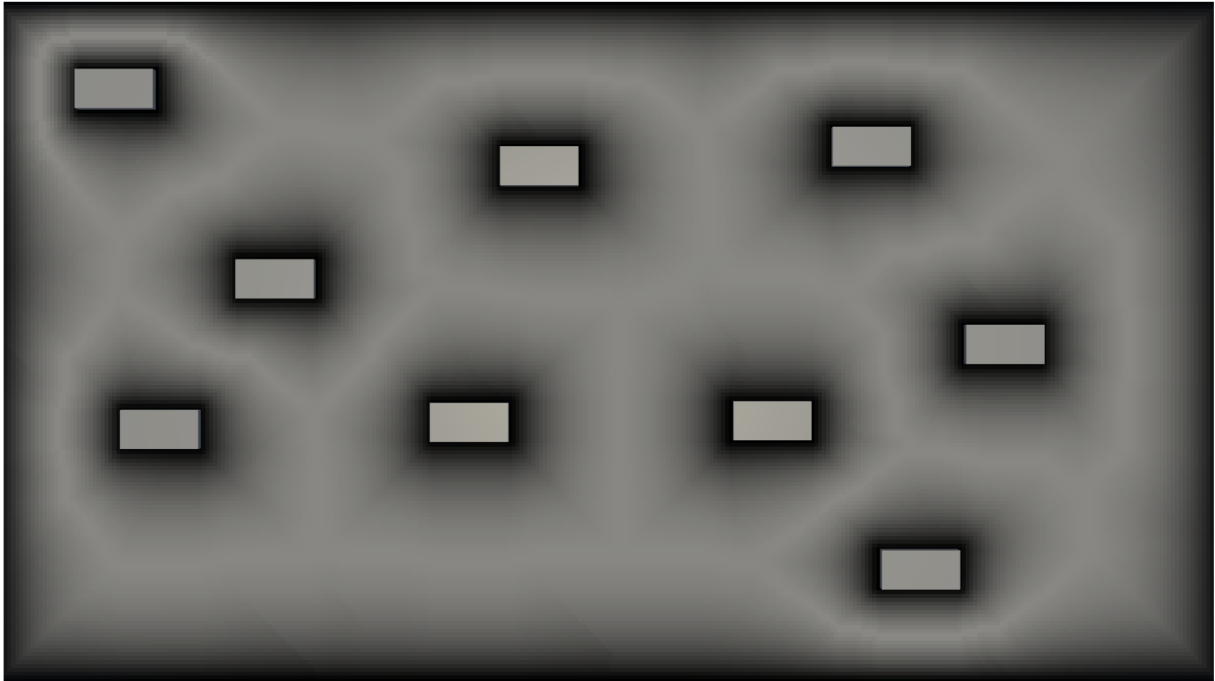
Fonte: Autoria própria (2024).

A Figura 20 apresenta o Campo de Voronoi em tons de cinza, onde os valores de  $\rho_V$  variam de 0 a 1. Neste caso, o valor 0 (representado em branco) corresponde às células de borda das regiões de influência, enquanto o valor 1 (representado em preto) indica as células próximas aos obstáculos. Qualquer valor intermediário de  $\rho_V$  é exibido em um tom de cinza proporcional à sua distância relativa aos obstáculos e às bordas, criando uma transição suave entre as regiões.

Para implementar o Campo de Voronoi, utilizou-se o Algoritmo 2. Esta lógica está encapsulada na classe `VoronoiField` (Anexo J), que organiza a propagação dos valores do campo, enquanto cada célula da grade é representada por uma instância de `VoronoiFieldCell` (Anexo K). A grade foi configurada de forma similar à descrita na Seção 4.3 (`Grid<VoronoiFieldCell>`), onde a `VoronoiFieldCell` armazena o valor de  $\rho_V$  e a distância de cada célula aos obstáculos mais próximos. Essa estrutura permite a representação eficiente do campo e facilita a adaptação dos parâmetros para ajustar a intensidade da influência dos obstáculos.

Os resultados obtidos confirmam que a implementação foi bem-sucedida na representação das áreas de influência dos obstáculos e na criação de um Campo de Voronoi que reflete gradualmente a proximidade dos obstáculos. Além disso, é possível ajustar os parâmetros da fórmula do Campo de Voronoi, caso seja necessário adaptar a intensidade do campo de acordo com diferentes requisitos. Nesta configuração, foram utilizados os parâmetros  $\alpha = 10$  e  $d_O^{\max} = 50$ , valores que se mostraram eficazes para gerar as transições desejadas no campo.

Figura 20 – Campo de Voronoi



Fonte: Autoria própria (2024).

Adicionalmente, o valor de  $\rho_V$  calculado no Campo de Voronoi foi utilizado como uma heurística adicional no algoritmo Hybrid A\* implementado neste trabalho. Essa heurística contribui para melhorar a eficiência do algoritmo ao penalizar regiões próximas a obstáculos, guiando o veículo por áreas com menor valor de  $\rho_V$ , o que representa zonas mais seguras e distantes de barreiras físicas. Assim, o Campo de Voronoi não só define áreas de influência dos obstáculos, mas também ajuda a evitar colisões de forma antecipada ao priorizar rotas mais distantes dos mesmos.

## 4.6 Hybrid A\*

A implementação do algoritmo Hybrid A\* neste trabalho utiliza as classes **HybridAStar** (Anexo N) e **Node** (Anexo M), sendo que a lógica do algoritmo está encapsulada na classe **HybridAStar**, conforme descrito no Algoritmo 4. A classe **Node** representa o estado de cada nó na busca, armazenando informações sobre a posição da roda traseira, o ângulo de direção, o custo acumulado e a heurística de cada movimento. Para gerenciar os nós a serem processados, também foi utilizada a classe **PriorityQueue**, da mesma forma que na implementação do algoritmo **AStar**, garantindo a expansão dos nós de menor custo total  $f$  primeiro, e, assim, melhorando a eficiência da busca.

Esta implementação segue a estrutura descrita na Subseção 3.2.3, na qual a grade inicial é criada com células do tipo **Cell**. A partir dessa grade, são geradas outras três grades especializadas com células do tipo **FlowFieldCell**, **AStarCell** e **VoronoiFieldCell**,

executando seus algoritmos específicos para gerar valores de heurística que serão usados pela classe **HybridAStar**. Além disso, a distância euclidiana é calculada e utilizada como uma heurística adicional. Assim, a classe **HybridAStar** recebe quatro matrizes distintas para serem aplicadas como heurísticas.

Durante a expansão dos nós, a classe **CarModel** é utilizada para calcular a nova posição e o ângulo do veículo, conforme descrito anteriormente, com uma distância de expansão de 0,7 metros — ligeiramente maior que o tamanho de célula de 0,5 metros, permitindo ajustes conforme necessário. A heurística aplicada segue a Equação 3.1, escolhida por meio de testes manuais que demonstraram ser a mais eficaz em diversos cenários, refletindo a dependência do algoritmo Hybrid A\* em uma boa heurística, que pode apresentar resultados variados conforme o contexto.

Além da expansão padrão de nós, foi implementada a Expansão Analítica para calcular a trajetória ideal entre o nó atual e o objetivo, utilizando a função **TryAnalyticalExpansion**. Esta função emprega o método de curvas de Reeds-Shepp, conforme descrito na Seção 2.5 e baseado na implementação disponível em <https://github.com/nathanlct/reeds-shepp-curves/tree/master>. Através da classe **ReedsShepp** (Anexo L), é gerada uma rota otimizada entre a posição atual e o destino, levando em conta a orientação do veículo em ambos os pontos.

Caso uma trajetória ideal seja encontrada, a função utiliza a classe **PathBuilder** para gerar pontos de passagem (ou *waypoints*) e ângulos intermediários, aplicando uma discretização de 0,35 metros. Essa discretização permite a verificação de colisões ao longo do trajeto, garantindo que a trajetória proposta seja viável e respeite as limitações físicas do ambiente.

Para assegurar que o veículo não colida com obstáculos ao longo do caminho, a Expansão Analítica inclui uma verificação de colisão detalhada para cada *waypoint*. A função calcula a posição central do veículo e os vértices de um retângulo que representa o contorno do veículo, com base em seu comprimento e largura. Em seguida, o módulo **Physics** do Unity é aplicado, utilizando as funções **Linecast** e **CheckSphere**. A função **Linecast** verifica cada par de vértices adjacentes do retângulo, simulando as bordas do veículo, enquanto a função **CheckSphere** é aplicada em cada vértice para detectar a presença de obstáculos próximos a esses pontos específicos. Esse processo de verificação eleva ligeiramente o contorno para evitar interferências com o solo e analisa tanto os segmentos do contorno quanto os vértices do veículo, identificando possíveis colisões com obstáculos ao longo do trajeto.

Se algum dos segmentos entre os vértices interceptar um obstáculo, a função **TryAnalyticalExpansion** retorna um valor nulo para o trajeto, indicando que a rota direta para o objetivo é inviável, e a busca continua. Esta abordagem de Expansão Analítica é essencial para otimizar o desempenho do algoritmo, pois permite que o Hybrid A\* busque

uma rota direta para o objetivo sempre que possível, reduzindo o tempo de cálculo e a complexidade do caminho, desde que a trajetória esteja livre de colisões.

Apesar de a Expansão Analítica em cada expansão de nó poder aumentar o custo computacional, principalmente pela necessidade de verificar colisões em cada vértice ao longo de toda a trajetória gerada pelas curvas de Reeds-Shepp, essa abordagem não apresenta problemas para o escopo deste trabalho, cujo foco não é o tempo de execução. Além disso, seria possível explorar outras trajetórias viáveis, não necessariamente a trajetória ótima, bem como atribuir pesos às trajetórias, penalizando, por exemplo, caminhos em que o veículo percorre grandes distâncias em marcha à ré. No entanto, tais ajustes não foram considerados neste trabalho.

A seguir, são apresentados os resultados da aplicação do algoritmo Hybrid A\* em diferentes cenários de teste, demonstrando sua capacidade de navegação e planejamento em espaços restritos e com obstáculos variados. Em todas as figuras, os nós onde o veículo se desloca para frente são representados em ciano, enquanto os nós onde a movimentação ocorre em marcha à ré aparecem em amarelo. A trajetória final, conectando a posição inicial ao destino, é indicada em laranja. Além disso, a posição do veículo é ilustrada a cada 0,35 metros ao longo do trajeto por um retângulo azul, que representa seu contorno, orientação e posição a intervalos regulares.

O primeiro cenário, representado nas Figuras 21 e 22, ilustra uma situação de estacionamento onde o veículo precisa manobrar cuidadosamente para se ajustar ao espaço disponível. A Figura 21 apresenta os nós expandidos pelo algoritmo Hybrid A\* com a Expansão Analítica, permitindo uma verificação detalhada do contorno do veículo em cada posição. Na Figura 22, observa-se apenas a trajetória final encontrada, evidenciando que o algoritmo conseguiu planejar um caminho viável e eficiente, sem colisões.

A análise desse cenário destaca a importância da configuração correta das heurísticas utilizadas. Durante os testes, observou-se que qualquer variação nos parâmetros da heurística influenciou diretamente a trajetória planejada, impactando tanto a segurança quanto a eficiência da navegação. A combinação das heurísticas selecionadas permitiu ao algoritmo priorizar caminhos mais próximos do objetivo, ao mesmo tempo em que mantinha uma margem de segurança em relação aos obstáculos. Contudo, parâmetros inadequados poderiam gerar expansões de nós desnecessárias e trajetórias menos eficientes, eventualmente levando o veículo a se aproximar perigosamente dos obstáculos. Esse cenário demonstra que um ajuste cuidadoso das heurísticas é fundamental para alcançar um caminho otimizado, seguro e adaptado ao ambiente de estacionamento.

O segundo cenário, representado nas Figuras 23 e 24 (sem Expansão Analítica) e nas Figuras 25 e 26 (com Expansão Analítica), o veículo deve alcançar um objetivo dentro de uma área altamente confinada, o que exige manobras complexas e precisas. Nas Figuras 23 e 24, sem a Expansão Analítica, é possível observar um grande número de nós

expandidos ao longo do caminho, indicando que o algoritmo expandiu muitos nós devido à dificuldade em manobrar no espaço restrito, ainda que tenha conseguido encontrar uma trajetória.

Quando a Expansão Analítica é ativada, como mostrado nas Figuras 25 e 26, observa-se uma redução significativa no número de nós expandidos e uma trajetória mais otimizada. A Expansão Analítica possibilitou uma análise mais detalhada das posições ao redor do veículo, permitindo ao algoritmo adaptar-se melhor ao espaço confinado e reduzir o custo de computação associado. Esse cenário demonstra que a Expansão Analítica pode ser essencial em ambientes confinados, reduzindo expansões desnecessárias e melhorando a eficiência do planejamento de trajetória.

O terceiro cenário, representado nas Figuras 27 e 28, coloca o veículo em uma trajetória complexa, na qual ele precisa atravessar uma área com múltiplos obstáculos antes de chegar a um ponto de estacionamento estreito. A Figura 27 exibe os nós expandidos ao longo do caminho, enquanto a Figura 28 mostra apenas a trajetória final. Neste caso, a Expansão Analítica ajudou o veículo a manter uma trajetória segura e eficiente ao manobrar entre os obstáculos, ajustando-se gradualmente ao espaço de estacionamento final.

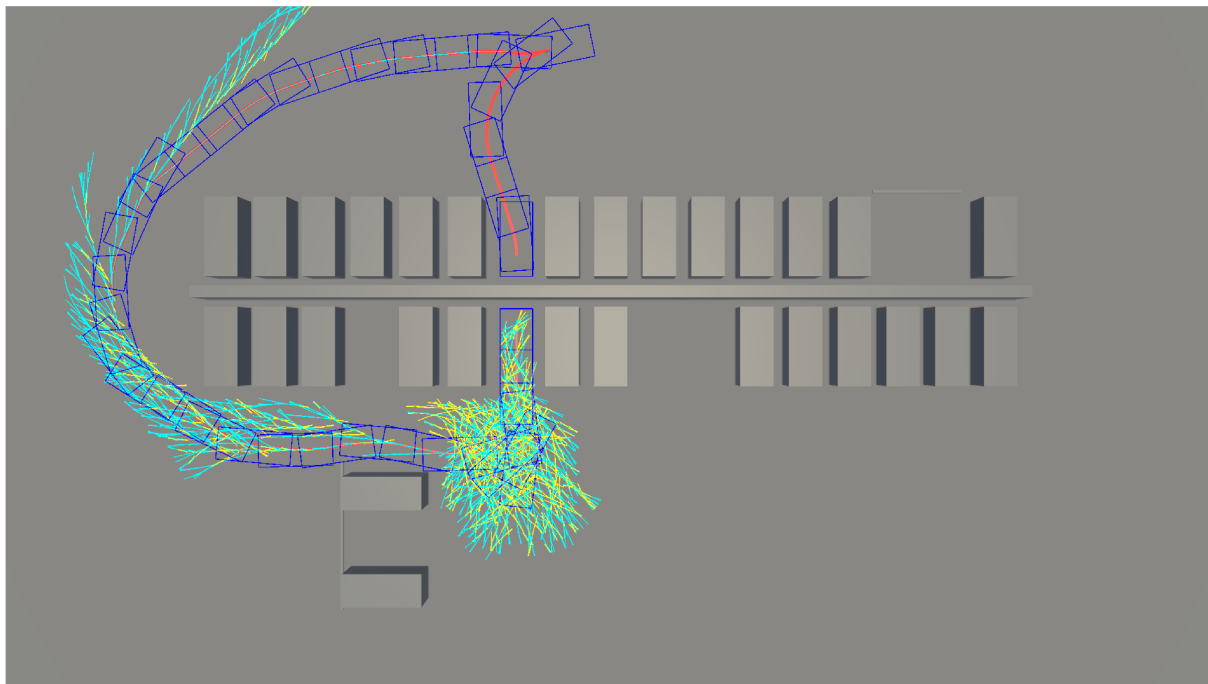
Neste cenário, as heurísticas foram fundamentais para guiar o veículo através da área de obstáculos sem desviar para caminhos ineficientes. A correta configuração dos parâmetros da heurística, em combinação com a Expansão Analítica, permitiu ao algoritmo priorizar trajetórias mais seguras e menos suscetíveis a colisões. Esse cenário evidencia a capacidade do Hybrid A\* em lidar com ambientes dinâmicos e desafiadores, desde que a heurística seja ajustada para balancear a eficiência da trajetória e a segurança do veículo.

Esses resultados comprovam a eficiência do algoritmo Hybrid A\* com Expansão Analítica em cenários desafiadores, proporcionando trajetórias precisas e otimizadas em espaços confinados e com obstáculos. Embora técnicas de suavização de trajetória, como discutido anteriormente, pudessem melhorar a qualidade do caminho gerado e até ajustar trajetórias onde ocorreram colisões, não foi necessário aplicá-las para alcançar bons resultados nos testes realizados. A abordagem mostrou-se suficiente para gerar caminhos viáveis e seguros, mesmo em situações complexas, sem a necessidade de suavização adicional.

A escolha da heurística se mostrou essencial para o desempenho do algoritmo. Durante os testes, observou-se que qualquer variação nos parâmetros da heurística afetava significativamente os resultados, muitas vezes alterando a trajetória de forma inesperada. A heurística baseada na Equação 3.1 foi a que apresentou o melhor desempenho em uma variedade de cenários, garantindo trajetórias seguras e eficientes. No entanto, mudanças nos parâmetros da heurística ou na configuração do Campo de Voronoi também impactaram drasticamente o caminho final, podendo até resultar em colisões ao longo da trajetória. Assim, o ajuste cuidadoso da heurística e dos parâmetros de geração do Campo de Voronoi

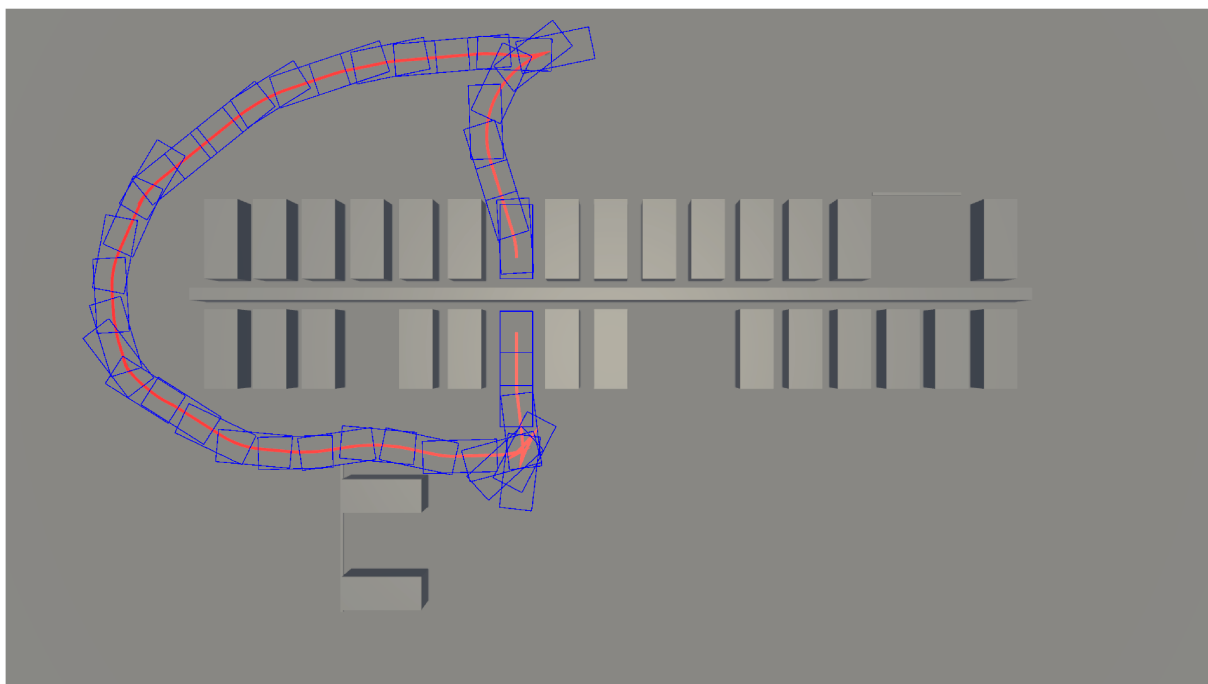
foi crucial para assegurar trajetórias viáveis e evitar erros de navegação em ambientes complexos.

Figura 21 – Cenário de estacionamento com Expansão Analítica, mostrando os nós expandidos ao longo do caminho.



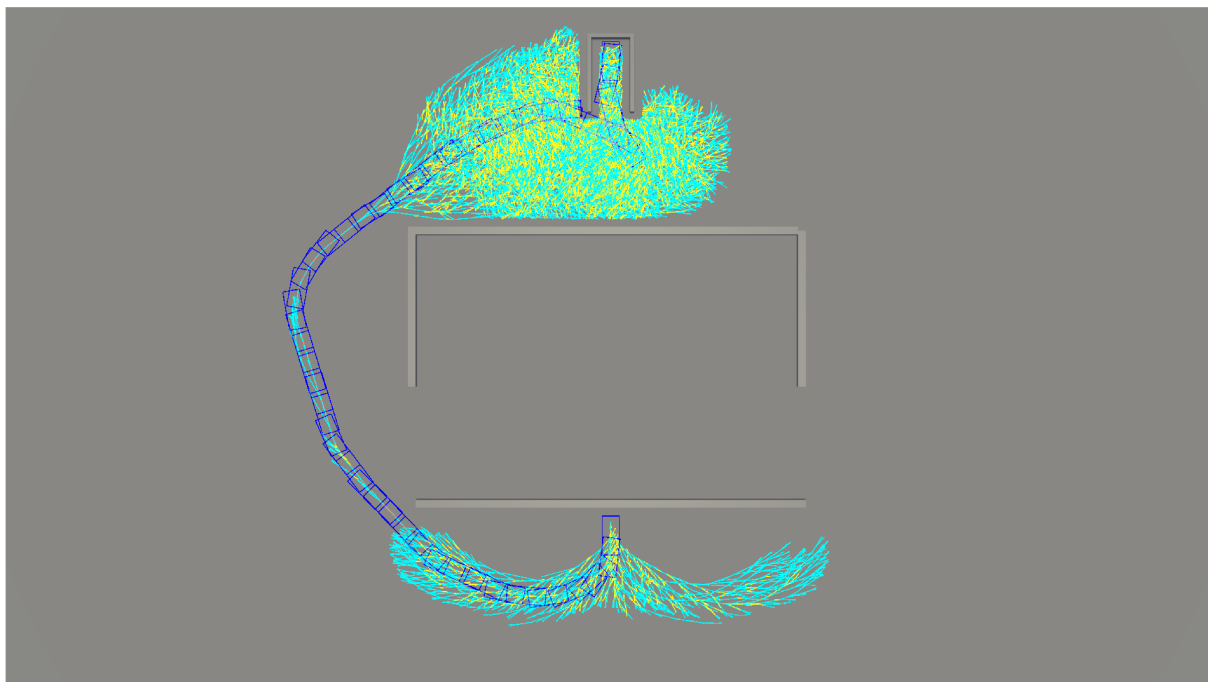
Fonte: Autoria própria (2024).

Figura 22 – Cenário de estacionamento com Expansão Analítica, exibindo apenas a trajetória final encontrada.



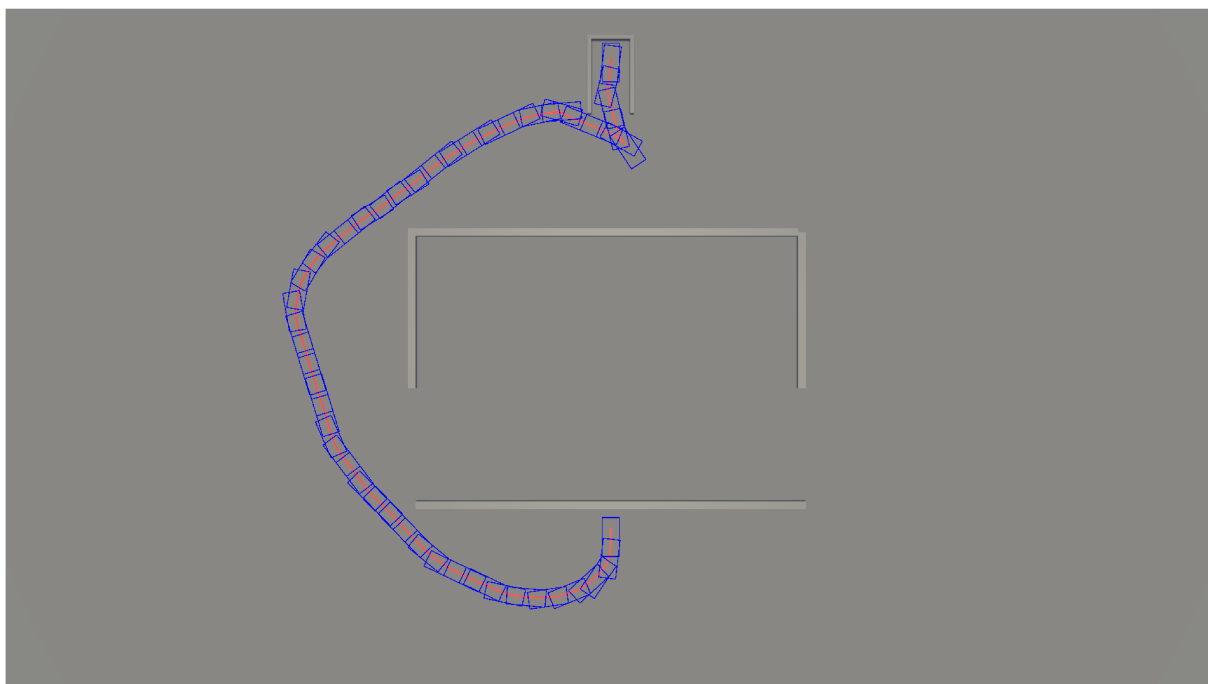
Fonte: Autoria própria (2024).

Figura 23 – Cenário de espaço confinado sem Expansão Analítica, mostrando os nós expandidos, evidenciando a grande quantidade de expansões necessárias.



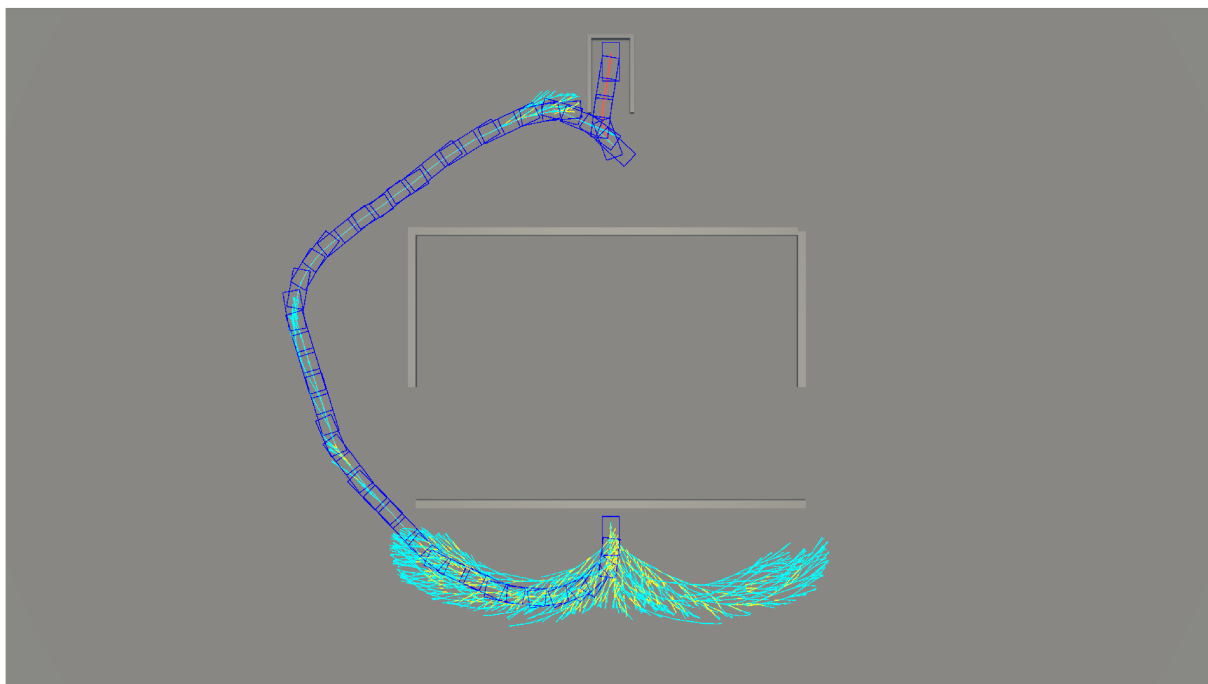
Fonte: Autoria própria (2024).

Figura 24 – Cenário de espaço confinado sem Expansão Analítica, exibindo apenas a trajetória final encontrada.



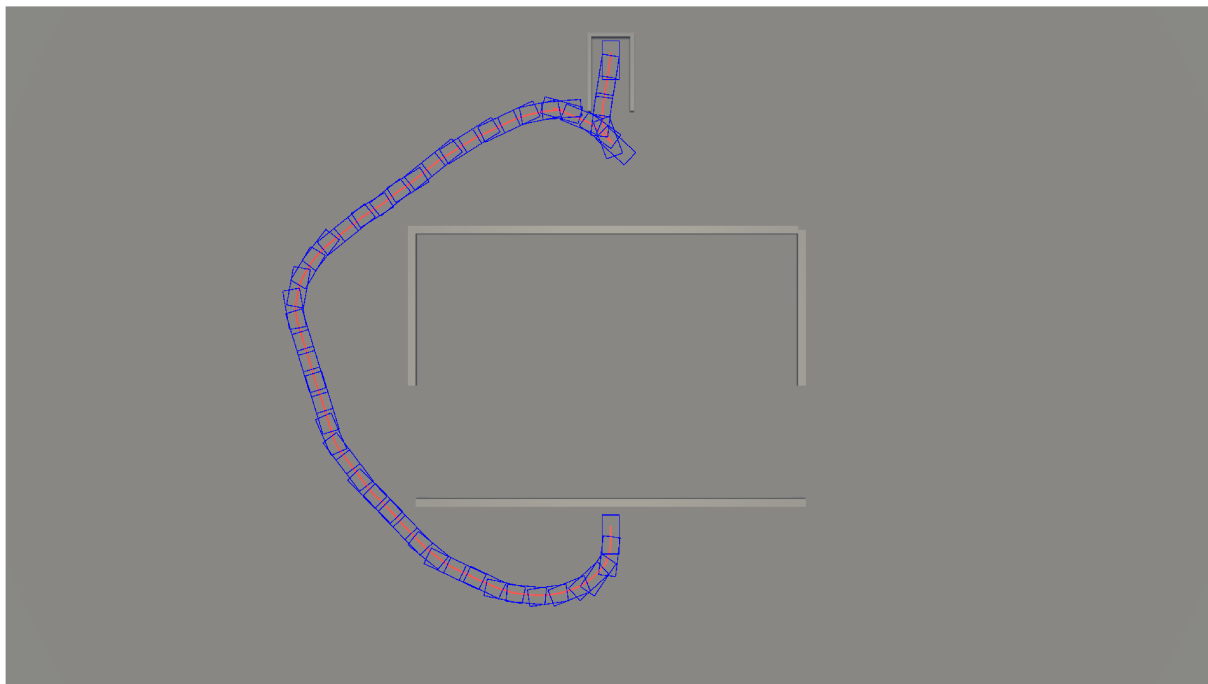
Fonte: Autoria própria (2024).

Figura 25 – Cenário de espaço confinado com Expansão Analítica, mostrando os nós expandidos, com uma redução significativa em relação ao caso sem Expansão Analítica.



Fonte: Autoria própria (2024).

Figura 26 – Cenário de espaço confinado com Expansão Analítica, exibindo apenas a trajetória final encontrada.



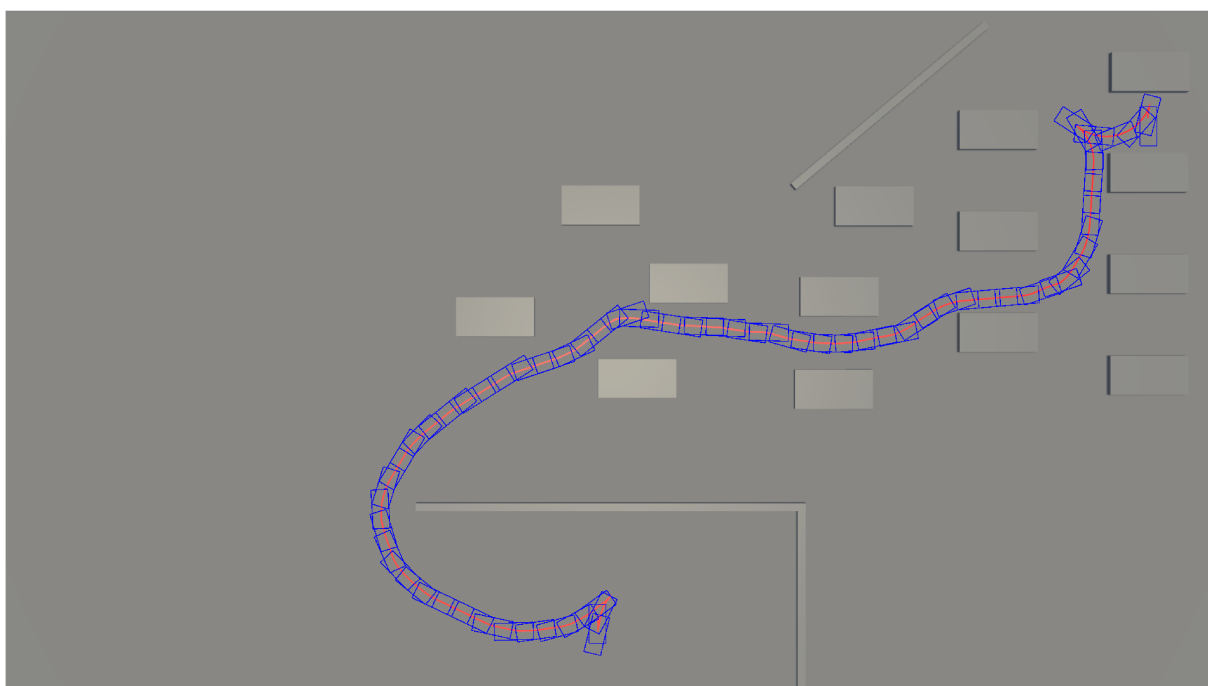
Fonte: Autoria própria (2024).

Figura 27 – Cenário complexo com múltiplos obstáculos e um ponto de estacionamento apertado, mostrando os nós expandidos ao longo da trajetória.



Fonte: Autoria própria (2024).

Figura 28 – Cenário complexo com múltiplos obstáculos e um ponto de estacionamento apertado, exibindo apenas a trajetória final encontrada.



Fonte: Autoria própria (2024).

## 5 Conclusão

Este trabalho apresentou a implementação e análise do algoritmo Hybrid A\* para o planejamento de trajetórias em veículos autônomos, utilizando a plataforma Unity como ambiente de simulação. Ao longo do estudo, foram exploradas as capacidades do Unity 3D para criar cenários realistas e complexos, permitindo a avaliação do desempenho do algoritmo em condições que se assemelham ao mundo real. O Hybrid A\*, como uma extensão do algoritmo A tradicional, demonstrou eficiência ao lidar com restrições não-holonômicas, que são essenciais para a definição de trajetórias seguras e viáveis para veículos autônomos.

Embora o desempenho do algoritmo dependa de uma heurística bem ajustada, os resultados obtidos confirmam a eficácia do Hybrid A\* em cenários com obstáculos estáticos e condições de navegação controladas. A criação de um ambiente virtual detalhado no Unity possibilitou a visualização e o teste das trajetórias, promovendo uma compreensão clara do comportamento do algoritmo e dos ajustes necessários para aprimorar a navegação autônoma.

Conclui-se que o Hybrid A\*, integrado ao Unity, apresenta um grande potencial para o desenvolvimento e teste de algoritmos de navegação autônoma, contribuindo significativamente para o avanço da pesquisa na área. Este trabalho também fornece uma documentação detalhada sobre a implementação do algoritmo, facilitando a replicação e a continuidade dos estudos.

Como sugestão para trabalhos futuros, recomenda-se a criação de um simulador interativo que permita a manipulação de obstáculos e do veículo, além de possibilitar a alteração dos pesos das heurísticas e a reorganização da equação heurística. Essa ferramenta pode facilitar a visualização das etapas do algoritmo e permitir a observação do veículo percorrendo a trajetória, contribuindo para uma melhor compreensão do desempenho do Hybrid A\* em diferentes cenários.

Além disso, é importante explorar ambientes dinâmicos que incluam obstáculos móveis e cenários mais complexos, a fim de aprimorar a robustez e a aplicabilidade do algoritmo em situações reais. A implementação de um modelo cinemático que represente de forma mais fiel um veículo real, juntamente com a avaliação de diversos tipos de controladores, pode enriquecer a análise da eficiência do algoritmo. Por fim, a busca por melhores métodos para determinar uma equação heurística representa uma área promissora para futuras pesquisas, contribuindo para o avanço do planejamento de trajetórias em veículos autônomos.

# Referências Bibliográficas

- ALBAHARI, J. *C# 12 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2023. ISBN 9781098147402. Disponível em: <<https://books.google.com.br/books?id=MuTiEAAAQBAJ>>. Citado 2 vezes nas páginas 28 e 50.
- AURENHAMMER, F. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 23, n. 3, p. 345–405, 1991. Citado na página 20.
- CHOSSET, H. et al. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005. (Intelligent Robotics and Autonomous Agents series). ISBN 9780262033275. Disponível em: <<https://books.google.com.br/books?id=S3biKR2li-QC>>. Citado 2 vezes nas páginas 22 e 35.
- COMMITTEE, O.-R. A. D. O. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. [S.l.], 2021. Disponível em: <[https://doi.org/10.4271/J3016\\_202104](https://doi.org/10.4271/J3016_202104)>. Citado na página 11.
- CORMEN, T. et al. *Introduction to Algorithms, Fourth Edition*. MIT Press, 2022. ISBN 9780262046305. Disponível em: <<https://books.google.com.br/books?id=drZNEAAAQBAJ>>. Citado na página 17.
- DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*, Springer, v. 1, n. 1, p. 269–271, 1959. Citado 2 vezes nas páginas 32 e 33.
- DOLGOV, D. et al. Practical search techniques in path planning for autonomous driving. *Ann Arbor*, v. 1001, n. 48105, p. 18–80, 2008. Citado 6 vezes nas páginas 21, 36, 37, 39, 40 e 41.
- DUBINS, L. E. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of mathematics*, JSTOR, v. 79, n. 3, p. 497–516, 1957. Citado na página 27.
- GONZÁLEZ, D. et al. A review of motion planning techniques for automated vehicles. *IEEE Transactions on intelligent transportation systems*, IEEE, v. 17, n. 4, p. 1135–1145, 2015. Citado na página 12.
- HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, IEEE, v. 4, n. 2, p. 100–107, 1968. Citado na página 32.
- JULIANI, A. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018. Citado na página 12.
- KNUTH, D. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Pearson Education, 1997. ISBN 9780321635747. Disponível em: <<https://books.google.com.br/books?id=x9AsAwAAQBAJ>>. Citado na página 18.

KURZER, K. *Path Planning in Unstructured Environments: A Real-time Hybrid A\* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle*. 63 p. Dissertação (Mestrado) — KTH, Integrated Transport Research Lab, ITRL, 2016. Citado 3 vezes nas páginas 41, 42 e 43.

LATOMBE, J. *Robot Motion Planning*. Springer US, 2012. (The Springer International Series in Engineering and Computer Science). ISBN 9781461540229. Disponível em: <<https://books.google.com.br/books?id=nQ7aBwAAQBAJ>>. Citado na página 34.

LAVALLE, S. *Planning Algorithms*. Cambridge University Press, 2006. ISBN 9781139455176. Disponível em: <<https://books.google.com.br/books?id=Clg8SWNMSRAC>>. Citado 2 vezes nas páginas 15 e 34.

MONTEMERLO, M. et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, Wiley Online Library, v. 25, n. 9, p. 569–597, 2008. Citado 3 vezes nas páginas 12, 33 e 36.

PADEN, B. et al. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, IEEE, v. 1, n. 1, p. 33–55, 2016. Citado na página 11.

POPP, K. et al. *Ground Vehicle Dynamics*. Springer Berlin Heidelberg, 2010. ISBN 9783540685531. Disponível em: <<https://books.google.com.br/books?id=tV6OSaFfqPgC>>. Citado na página 16.

REEDS, J.; SHEPP, L. Optimal paths for a car that goes both forwards and backwards. *Pacific journal of mathematics*, Mathematical Sciences Publishers, v. 145, n. 2, p. 367–393, 1990. Citado na página 27.

RUSSELL, S.; NORVIG, P.; DAVIS, E. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010. (Prentice Hall series in artificial intelligence). ISBN 9780136042594. Disponível em: <<https://books.google.com.br/books?id=8jZBksh-bUMC>>. Citado na página 33.

SINGH, S. *Critical reasons for crashes investigated in the national motor vehicle crash causation survey*. [S.l.], 2015. Citado na página 11.

SUSSMANN, H. J.; TANG, G. *Shortest paths for the Reeds-Shepp car: a worked out example of the use of geometric techniques in nonlinear optimal control*. [S.l.]: SYCON, 1991. Citado na página 27.

SVIATOV, K. et al. Approach of trajectory generation based on waypoint information for a highly automated vehicle. Preprints, 2023. Citado na página 28.

TAKAHASHI, O.; SCHILLING, R. J. Motion planning in a plane using generalized voronoi diagrams. *IEEE Transactions on robotics and automation*, IEEE, v. 5, n. 2, p. 143–150, 1989. Citado na página 21.

TESLA, I. *Model S Owner's Manual*. [S.l.], 2021. Software version: 2024.38, North America. Disponível em: <[https://www.tesla.com/ownersmanual/models/en\\_us/Owners\\_Manual.pdf](https://www.tesla.com/ownersmanual/models/en_us/Owners_Manual.pdf)>. Citado na página 47.

UNITY-TECHNOLOGIES. *Unity Real-Time Development Platform | 3D, 2D, VR and AR Engine*. 2024. Disponível em: <<https://unity.com/>>. Citado 2 vezes nas páginas 11 e 28.

XU, W. et al. A vehicle model for micro-traffic simulation in dynamic urban scenarios. In: IEEE. *2011 IEEE international conference on robotics and automation*. [S.l.], 2011. p. 2267–2274. Citado na página 16.

## ANEXO A – Classe Grid

```

1  using System;
2  using UnityEngine;
3
4  public class Grid<T> where T : Cell
5  {
6      public Vector3 Size { get; }
7
8      public Vector3 Center { get; }
9
10     public float CellSize { get; }
11
12     public LayerMask UnwalkableMask;
13
14     private Func<Vector3, bool, int, int, T> _cellFactory;
15
16     public T[,] Cells;
17
18     public bool[,] Obstacles;
19
20     public int Width { get; }
21
22     public int Height { get; }
23
24     public int MaxSize => Width * Height;
25
26     public Grid(Vector3 size, Vector3 center, float cellSize, LayerMask
    ↪  unwalkableMask,
27         Func<Vector3, bool, int, int, T> cellFactory, bool[,] obstacles = null)
28     {
29         Size = size;
30         Center = center;
31         CellSize = cellSize;
32         UnwalkableMask = unwalkableMask;
33         _cellFactory = cellFactory;
34
35         Width = Mathf.RoundToInt(size.x / cellSize);
36         Height = Mathf.RoundToInt(size.z / cellSize);
37
38         Cells = new T[Width, Height];
39         Obstacles = new bool[Width, Height];
40
41         for (var i = 0; i < Width; i++)
42         {
43             for (var j = 0; j < Height; j++)
44             {
45                 var position = new Vector3(
46                     i * CellSize - (Width - 1) / 2f * CellSize + Center.x,
47                     Center.y,
48                     j * CellSize - (Height - 1) / 2f * CellSize + Center.z
49                 );
50
51                 var isWalkable = obstacles != null ? !obstacles[i, j] :
    ↪  !CheckCollision(position);
52
53                 Cells[i, j] = _cellFactory(position, isWalkable, i, j);
54
55                 Obstacles[i, j] = !isWalkable;
56             }
57         }
58     }
59
60     public bool CheckCollision(Vector3 position)
61     {
62         const float delta = 0.5945f + 0.1f;
63         //const float delta = 0;
64     }

```

```
65         var collided =
66             Physics.CheckBox(position,
67                 new Vector3(CellSize / 2f + delta, 0.1f, CellSize / 2f + delta),
68                 Quaternion.identity,
69                 UnwalkableMask);
70
71         return collided;
72     }
73
74     public bool TryGetCell(int x, int y, out T cell)
75     {
76         if (x < 0 || x >= Width || y < 0 || y >= Height)
77         {
78             cell = null;
79             return false;
80         }
81
82         cell = Cells[x, y];
83         return true;
84     }
85
86     public bool TryGetCellFromWorldPosition(Vector3 position, out T cell)
87     {
88         var relativePosition = position - Center;
89
90         var x = Mathf.FloorToInt((relativePosition.x + Width / 2f * CellSize) /
91             ⇨ CellSize);
92         var y = Mathf.FloorToInt((relativePosition.z + Height / 2f * CellSize) /
93             ⇨ CellSize);
94
95         return TryGetCell(x, y, out cell);
96     }
97
98     public bool IsPositionWalkable(Vector3 position)
99     {
100         if (!TryGetCellFromWorldPosition(position, out var cell))
101         {
102             return false;
103         }
104
105         return cell.IsWalkable;
106     }
107 }
```

## ANEXO B – Classe Cell

```
1  using UnityEngine;
2
3  public class Cell
4  {
5      public Vector3 Center;
6      public bool IsWalkable;
7      public Vector2Int GridPosition;
8
9      public Cell(Vector3 center, bool isWalkable, int x, int y)
10     {
11         Center = center;
12         IsWalkable = isWalkable;
13         GridPosition = new Vector2Int(x, y);
14     }
15 }
```

## ANEXO C – Classe Movement

```
1 public class Movement
2 {
3     public enum Steering
4     {
5         Left = -1,
6         Straight = 0,
7         Right = 1
8     };
9
10    public enum Gear
11    {
12        Backward = -1,
13        Forward = 1
14    };
15
16    public float Distance { get; set; }
17    public Steering SteeringVal { get; set; }
18    public Gear GearVal { get; set; }
19
20    public Movement(float distance, Steering steering, Gear gear)
21    {
22        Distance = distance >= 0 ? distance : -distance;
23        SteeringVal = steering;
24        GearVal = distance >= 0 ? gear : (Gear)(- (int)gear);
25    }
26
27    public void ReverseSteering()
28    {
29        SteeringVal = (Steering)(- (int)SteeringVal);
30    }
31
32    public void ReverseGear()
33    {
34        GearVal = (Gear)(- (int)GearVal);
35    }
36 }
```

## ANEXO D – Classe CarModel

```

1  using UnityEngine;
2
3  public class CarModel
4  {
5      public static (Vector3, float) CalculateMovement(Vector3 rearWheelPosition, float
        ↳ headingAngle, Movement movement)
6      {
7          var headingRadians = headingAngle * Mathf.Deg2Rad;
8          var newRearWheelPos = rearWheelPosition;
9          var newHeadingAngle = headingAngle;
10
11         switch (movement.SteeringVal)
12         {
13             case Movement.Steering.Left:
14             case Movement.Steering.Right:
15                 var centerX = rearWheelPosition.x + (movement.SteeringVal ==
        ↳ Movement.Steering.Left ? -1 : 1) *
16                     Mathf.Cos(headingRadians) * Constants.CarTurningRadius;
17                 var centerZ = rearWheelPosition.z + (movement.SteeringVal ==
        ↳ Movement.Steering.Left ? 1 : -1) *
18                     Mathf.Sin(headingRadians) * Constants.CarTurningRadius;
19
20                 var angleChange = movement.Distance /
        ↳ Constants.CarTurningCircumference * 2 * Mathf.PI;
21
22                 angleChange *= movement.GearVal == Movement.Gear.Forward ? 1 : -1;
23                 angleChange *= movement.SteeringVal == Movement.Steering.Left ? -1 :
        ↳ 1;
24
25                 var newAngle = headingRadians + angleChange;
26
27                 newRearWheelPos.x = centerX + (movement.SteeringVal ==
        ↳ Movement.Steering.Left ? 1 : -1) *
28                     Constants.CarTurningRadius * Mathf.Cos(newAngle);
29                 newRearWheelPos.z = centerZ + (movement.SteeringVal ==
        ↳ Movement.Steering.Left ? -1 : 1) *
30                     Constants.CarTurningRadius * Mathf.Sin(newAngle);
31
32                 newHeadingAngle = Utils.WrapAngle((newAngle * Mathf.Rad2Deg));
33
34                 break;
35
36             case Movement.Steering.Straight:
37                 var distance = movement.Distance * (movement.GearVal ==
        ↳ Movement.Gear.Forward ? 1 : -1);
38
39                 newRearWheelPos.x += distance * Mathf.Sin(headingRadians);
40                 newRearWheelPos.z += distance * Mathf.Cos(headingRadians);
41
42                 break;
43         }
44
45         return (newRearWheelPos, newHeadingAngle);
46     }
47 }

```

# ANEXO E – Classe PriorityQueue

```

1  using System;
2
3  public class PriorityQueue<T> where T : IComparable<T>
4  {
5      private T[] items;
6      private int count;
7
8      public PriorityQueue(int maxSize)
9      {
10         items = new T[maxSize];
11         count = 0;
12     }
13
14     public int Count => count;
15
16     public void Enqueue(T item)
17     {
18         if (count == items.Length)
19             throw new InvalidOperationException("Queue is full");
20
21         items[count] = item;
22         HeapifyUp(count);
23         count++;
24     }
25
26     public T Dequeue()
27     {
28         if (count == 0)
29             throw new InvalidOperationException("Queue is empty");
30
31         var firstItem = items[0];
32         count--;
33         items[0] = items[count];
34
35         if (count > 0)
36             HeapifyDown(0);
37
38         return firstItem;
39     }
40
41     public bool Contains(T item)
42     {
43         for (var i = 0; i < count; i++)
44         {
45             if (items[i].Equals(item))
46                 return true;
47         }
48
49         return false;
50     }
51
52     private void HeapifyUp(int index)
53     {
54         while (index > 0)
55         {
56             var parentIndex = (index - 1) / 2;
57             if (items[index].CompareTo(items[parentIndex]) <= 0)
58                 break;
59
60             Swap(index, parentIndex);
61             index = parentIndex;
62         }
63     }
64
65     private void HeapifyDown(int index)

```

```
66     {
67         while (true)
68         {
69             var leftChildIndex = 2 * index + 1;
70             var rightChildIndex = 2 * index + 2;
71             var largestIndex = index;
72
73             if (leftChildIndex < count &&
74                 ⇨ items[leftChildIndex].CompareTo(items[largestIndex]) > 0)
75                 largestIndex = leftChildIndex;
76
77             if (rightChildIndex < count &&
78                 ⇨ items[rightChildIndex].CompareTo(items[largestIndex]) > 0)
79                 largestIndex = rightChildIndex;
80
81             if (largestIndex == index)
82                 break;
83
84             Swap(index, largestIndex);
85             index = largestIndex;
86         }
87     }
88
89     private void Swap(int indexA, int indexB)
90     {
91         (items[indexA], items[indexB]) = (items[indexB], items[indexA]);
92     }
93 }
```

## ANEXO F – Classe AStarCell

```

1  using System;
2  using UnityEngine;
3
4  public class AStarCell : Cell, IComparable<AStarCell>
5  {
6      public float gCost { get; private set; }
7
8      public float hCost { get; private set; }
9
10     public float fCost => gCost + hCost;
11
12     public AStarCell Parent { get; private set; }
13
14     public AStarCell(Vector3 center, bool isWalkable, int x, int y) : base(center,
15     ↪ isWalkable, x, y)
16     {
17     }
18
19     public void SetCosts(float g, float h)
20     {
21         gCost = g;
22         hCost = h;
23     }
24
25     public void SetParent(AStarCell parent)
26     {
27         Parent = parent;
28     }
29
30     public void Reset()
31     {
32         gCost = 0;
33         hCost = 0;
34         Parent = null;
35     }
36
37     public int CompareTo(AStarCell other)
38     {
39         var compare = fCost.CompareTo(other.fCost);
40
41         if (compare == 0)
42         {
43             compare = hCost.CompareTo(other.hCost);
44         }
45
46         return -compare;
47     }
48 }

```

# ANEXO G – Classe AStar

```

1  using System.Collections.Generic;
2
3  public class AStar
4  {
5      public static List<AStarCell> Run(Grid<AStarCell> grid, AStarCell startCell,
6          ↪ AStarCell goalCell)
7      {
8          for (var i = 0; i < grid.Width; i++)
9          {
10             for (var j = 0; j < grid.Height; j++)
11             {
12                 var cell = grid.Cells[i, j];
13                 cell.Reset();
14             }
15         }
16
17         var openSet = new PriorityQueue<AStarCell>(grid.MaxSize);
18         var closedSet = new HashSet<AStarCell>();
19
20         openSet.Enqueue(startCell);
21
22         while (openSet.Count > 0)
23         {
24             var currentCell = openSet.Dequeue();
25
26             closedSet.Add(currentCell);
27
28             foreach (var neighbor in GetNeighboringCells(grid, currentCell))
29             {
30                 if (!neighbor.IsWalkable) continue;
31
32                 if (closedSet.Contains(neighbor)) continue;
33
34                 var distanceToNeighbor = (currentCell.Center -
35                     ↪ neighbor.Center).magnitude;
36
37                 var gCost = currentCell.gCost + distanceToNeighbor;
38
39                 if (openSet.Contains(neighbor) && gCost >= neighbor.gCost) continue;
40
41                 var distanceToGoal = (currentCell.Center - goalCell.Center).magnitude;
42
43                 neighbor.SetCosts(gCost, distanceToGoal);
44                 neighbor.SetParent(currentCell);
45
46                 if (!openSet.Contains(neighbor))
47                 {
48                     openSet.Enqueue(neighbor);
49                 }
50             }
51
52             return RetracePath(goalCell);
53         }
54
55         private static List<AStarCell> GetNeighboringCells(Grid<AStarCell> grid, Cell
56             ↪ cell)
57         {
58             var neighbors = new List<AStarCell>();
59
60             var directions = Utils.DirectionsWithCorners;
61
62             foreach (var direction in directions)
63             {

```

```
63         var x = cell.GridPosition.x + direction.x;
64         var y = cell.GridPosition.y + direction.y;
65
66         if (!grid.TryGetCell(x, y, out var neighbor)) continue;
67
68         if (!neighbor.IsWalkable) continue;
69
70         neighbors.Add(neighbor);
71     }
72
73     return neighbors;
74 }
75
76 private static List<AStarCell> RetracePath(AStarCell goalCell)
77 {
78     var path = new List<AStarCell>();
79
80     var currentCell = goalCell;
81
82     while (currentCell != null)
83     {
84         path.Add(currentCell);
85         currentCell = currentCell.Parent;
86     }
87
88     path.Reverse();
89
90     return path;
91 }
92 }
```

## ANEXO H – Classe FlowFieldCell

```

1  using System.Collections.Generic;
2  using UnityEngine;
3
4  public class FlowFieldCell : Cell
5  {
6      public float Cost { get; private set; }
7
8      public int Region { get; private set; }
9
10     public HashSet<FlowFieldCell> Neighbors { get; private set; } = new();
11
12     public HashSet<FlowFieldCell> ClosestStartCells { get; } = new();
13
14     public FlowFieldCell(Vector3 center, bool isWalkable, int x, int y) : base(center,
15     ⇨ isWalkable, x, y)
16     {
17         Region = -1;
18     }
19
20     public void SetCost(float cost)
21     {
22         Cost = cost;
23     }
24
25     public void SetRegion(int region)
26     {
27         Region = region;
28     }
29
30     public void SetNeighbors(HashSet<FlowFieldCell> neighbors)
31     {
32         Neighbors = neighbors;
33     }
34
35     public void AddClosestStartCell(FlowFieldCell cell)
36     {
37         ClosestStartCells.Add(cell);
38     }
39
40     public void ClearClosestStartCells()
41     {
42         ClosestStartCells.Clear();
43     }
44
45     public void Reset()
46     {
47         Cost = float.MaxValue;
48         Neighbors.Clear();
49         ClosestStartCells.Clear();
50     }
51 }

```

# ANEXO I – Classe FlowField

```

1  using System.Collections.Generic;
2
3  public class FlowField
4  {
5      public static void CalculateFlowField(Grid<FlowFieldCell> grid, List<FlowFieldCell>
        ↪ startCells)
6      {
7          for (var i = 0; i < grid.Width; i++)
8          {
9              for (var j = 0; j < grid.Height; j++)
10             {
11                 var cell = grid.Cells[i, j];
12
13                 cell.Reset();
14
15                 var neighbors = GetNeighboringCells(grid, cell);
16
17                 cell.SetNeighbors(neighbors);
18             }
19         }
20
21         var openSet = new Queue<FlowFieldCell>();
22         var closedSet = new HashSet<FlowFieldCell>();
23
24         foreach (var cell in startCells)
25         {
26             openSet.Enqueue(cell);
27
28             cell.SetCost(0);
29
30             cell.AddClosestStartCell(cell);
31
32             closedSet.Add(cell);
33         }
34
35         while (openSet.Count > 0)
36         {
37             var currentCell = openSet.Dequeue();
38
39             foreach (var neighbor in currentCell.Neighbors)
40             {
41                 var distance = (currentCell.Center - neighbor.Center).magnitude;
42
43                 var newCost = currentCell.Cost + distance;
44
45                 if (newCost > neighbor.Cost) continue;
46
47                 neighbor.SetCost(newCost);
48
49                 neighbor.SetRegion(currentCell.Region);
50
51                 if (!newCost.Equals(neighbor.Cost))
52                 {
53                     neighbor.ClearClosestStartCells();
54                 }
55
56                 foreach (var cell in currentCell.ClosestStartCells)
57                 {
58                     neighbor.AddClosestStartCell(cell);
59                 }
60
61                 if (closedSet.Contains(neighbor)) continue;
62
63                 openSet.Enqueue(neighbor);
64

```

```

65         closedSet.Add(neighbor);
66     }
67 }
68 }
69
70 private static HashSet<FlowFieldCell> GetNeighboringCells(Grid<FlowFieldCell> grid,
71 ↪ Cell cell,
72     bool includeCorners = true)
73 {
74     var neighbors = new HashSet<FlowFieldCell>();
75
76     var directions = includeCorners ? Utils.DirectionsWithCorners :
77 ↪     Utils.DirectionsWithoutCorners;
78
79     var hasAnyNeighborObstacle = false;
80
81     foreach (var direction in directions)
82     {
83         var x = cell.GridPosition.x + direction.x;
84         var y = cell.GridPosition.y + direction.y;
85
86         if (!grid.TryGetCell(x, y, out var neighbor)) continue;
87
88         if (!neighbor.IsWalkable)
89         {
90             hasAnyNeighborObstacle = true;
91             continue;
92         }
93
94         neighbors.Add(neighbor);
95     }
96
97     if (!includeCorners || !hasAnyNeighborObstacle) return neighbors;
98
99     RemoveInvalidCornerNeighbors(grid, cell, neighbors);
100
101     return neighbors;
102 }
103
104 private static void RemoveInvalidCornerNeighbors(Grid<FlowFieldCell> grid, Cell
105 ↪ cell,
106     HashSet<FlowFieldCell> neighbors)
107 {
108     foreach (var direction in Utils.CornerDirections)
109     {
110         var x = cell.GridPosition.x + direction.x;
111         var y = cell.GridPosition.y + direction.y;
112
113         if (!grid.TryGetCell(x, y, out var cornerNeighbor)) continue;
114
115         var x1 = cell.GridPosition.x + direction.x;
116         var y1 = cell.GridPosition.y;
117
118         var x2 = cell.GridPosition.x;
119         var y2 = cell.GridPosition.y + direction.y;
120
121         if (!grid.TryGetCell(x1, y1, out var neighbor1) || !grid.TryGetCell(x2, y2,
122 ↪     out var neighbor2)) continue;
123
124         if (!neighbor1.IsWalkable || !neighbor2.IsWalkable)
125         {
126             neighbors.Remove(cornerNeighbor);
127         }
128     }
129 }
130 }

```

## ANEXO J – Classe VoronoiFieldCell

```

1  using System.Collections.Generic;
2  using UnityEngine;
3
4  public class VoronoiFieldCell : Cell
5  {
6      public int Region { get; private set; }
7
8      public bool IsEdge { get; private set; }
9
10     public float ClosestObstacleDistance { get; private set; }
11
12     public Vector3 ClosestObstaclePosition { get; private set; }
13
14     public float ClosestEdgeDistance { get; private set; }
15
16     public Vector3 ClosestEdgePosition { get; private set; }
17
18     public float FieldValue { get; private set; }
19
20     public List<VoronoiFieldCell> ClosestObstacleCells { get; private set; }
21
22     public List<VoronoiFieldCell> ClosestEdgeCells { get; private set; }
23
24     public VoronoiFieldCell(Vector3 center, bool isWalkable, int x, int y) :
25     ↪ base(center, isWalkable, x, y)
26     {
27         Region = -1;
28         IsEdge = false;
29     }
30
31     public void SetRegion(int region)
32     {
33         Region = region;
34     }
35
36     public void SetIsEdge(bool isEdge)
37     {
38         IsEdge = isEdge;
39     }
40
41     public void SetClosestObstacleDistance(float distance)
42     {
43         ClosestObstacleDistance = distance;
44     }
45
46     public void SetClosestObstaclePosition(Vector3 position)
47     {
48         ClosestObstaclePosition = position;
49     }
50
51     public void SetFieldValue(float value)
52     {
53         FieldValue = value;
54     }
55
56     public void SetClosestObstacleCells(List<VoronoiFieldCell> cells)
57     {
58         ClosestObstacleCells = cells;
59     }
60
61     public void SetClosestEdgeDistance(float distance)
62     {
63         ClosestEdgeDistance = distance;
64     }
65
66     public void SetClosestEdgePosition(Vector3 position)

```

---

```
66     {
67         ClosestEdgePosition = position;
68     }
69
70     public void SetClosestEdgeCells(List<VoronoiFieldCell> cells)
71     {
72         ClosestEdgeCells = cells;
73     }
74 }
```

## ANEXO K – Classe VoronoiField

```

1  using System.Collections.Generic;
2
3  public class VoronoiField
4  {
5      public static void GenerateVoronoiField(Grid<VoronoiFieldCell> grid)
6      {
7          FindObstacleRegions(grid);
8
9          FindVoronoiRegions(grid);
10
11         FindVoronoiEdges(grid);
12
13         FindDistancesToEdges(grid);
14
15         for (var i = 0; i < grid.Width; i++)
16         {
17             for (var j = 0; j < grid.Height; j++)
18             {
19                 var currentCell = grid.Cells[i, j];
20
21                 float rho;
22
23                 if (!currentCell.IsWalkable)
24                 {
25                     rho = 1f;
26                 }
27                 else if (currentCell.IsEdge)
28                 {
29                     rho = 0;
30                 }
31                 else
32                 {
33                     var closestObstacleDistance = currentCell.ClosestObstacleDistance;
34                     var closestEdgeDistance = currentCell.ClosestEdgeDistance;
35
36                     rho = (Constants.VoronoiAlpha / (Constants.VoronoiAlpha +
37 ↪ closestObstacleDistance)) *
38 ↪ (closestEdgeDistance / (closestObstacleDistance +
39 ↪ closestEdgeDistance)) *
40 ↪ (((closestObstacleDistance -
41 ↪ Constants.VoronoiMaxObstacleDistance) *
42 ↪ (closestObstacleDistance -
43 ↪ Constants.VoronoiMaxObstacleDistance)) /
44 ↪ (Constants.VoronoiMaxObstacleDistance *
45 ↪ Constants.VoronoiMaxObstacleDistance));
46                 }
47
48                 currentCell.SetFieldValue(rho);
49             }
50         }
51     }
52
53     private static void FindObstacleRegions(Grid<VoronoiFieldCell> grid)
54     {
55         var region = 0;
56
57         for (var x = 0; x < grid.Width; x++)
58         {
59             for (var y = 0; y < grid.Height; y++)
60             {
61                 var cell = grid.Cells[x, y];
62
63                 if (cell.IsWalkable) continue;
64
65                 if (cell.Region != -1) continue;

```

```

61         FloodFillObstacle(grid, cell, region);
62     }
63     region++;
64 }
65 }
66 }
67 }
68
69 private static void FloodFillObstacle(Grid<VoronoiFieldCell> grid,
    ↪ VoronoiFieldCell startCell, int region)
70 {
71     var openSet = new Queue<VoronoiFieldCell>();
72     openSet.Enqueue(startCell);
73     while (openSet.Count > 0)
74     {
75         var currentCell = openSet.Dequeue();
76         currentCell.SetRegion(region);
77         foreach (var direction in Utils.DirectionsWithoutCorners)
78         {
79             var x = currentCell.GridPosition.x + direction.x;
80             var y = currentCell.GridPosition.y + direction.y;
81             if (!grid.TryGetCell(x, y, out var neighbor)) continue;
82             if (neighbor.IsWalkable) continue;
83             if (neighbor.Region != -1) continue;
84             if (openSet.Contains(neighbor)) continue;
85             openSet.Enqueue(neighbor);
86         }
87     }
88 }
89
90 private static void FindVoronoiRegions(Grid<VoronoiFieldCell> grid)
91 {
92     var flowFieldGrid = new Grid<FlowFieldCell>(grid.Size, grid.Center,
    ↪ grid.CellSize, grid.UnwalkableMask,
93     (center, _, x, y) => new FlowFieldCell(center, true, x, y),
    ↪ grid.Obstacles);
94
95     for (var x = 0; x < grid.Width; x++)
96     {
97         for (var y = 0; y < grid.Height; y++)
98         {
99             var flowFieldCell = flowFieldGrid.Cells[x, y];
100             var voronoiFieldCell = grid.Cells[x, y];
101             flowFieldCell.SetRegion(voronoiFieldCell.Region);
102         }
103     }
104
105     var startCells = new List<FlowFieldCell>();
106
107     for (var x = 0; x < grid.Width; x++)
108     {
109         for (var y = 0; y < grid.Height; y++)
110         {
111             var voronoiFieldCell = grid.Cells[x, y];
112             if (voronoiFieldCell.IsWalkable) continue;
113             var flowFieldCell = flowFieldGrid.Cells[x, y];
114             startCells.Add(flowFieldCell);
115         }
116     }
117 }

```

```

130     }
131
132     FlowField.CalculateFlowField(flowFieldGrid, startCells);
133
134     for (var x = 0; x < grid.Width; x++)
135     {
136         for (var y = 0; y < grid.Height; y++)
137         {
138             var voronoiFieldCell = grid.Cells[x, y];
139
140             var flowFieldCell = flowFieldGrid.Cells[x, y];
141
142             var closestObstacleDistance = flowFieldCell.Cost;
143
144             voronoiFieldCell.SetClosestObstacleDistance(closestObstacleDistance);
145             voronoiFieldCell.SetRegion(flowFieldCell.Region);
146         }
147     }
148 }
149
150 private static void FindVoronoiEdges(Grid<VoronoiFieldCell> grid)
151 {
152     for (var i = 0; i < grid.Width; i++)
153     {
154         for (var j = 0; j < grid.Height; j++)
155         {
156             var currentCell = grid.Cells[i, j];
157
158             var currentRegion = currentCell.Region;
159
160             foreach (var direction in Utils.DirectionsWithoutCorners)
161             {
162                 var x = i + direction.x;
163                 var y = j + direction.y;
164
165                 if (!grid.TryGetCell(x, y, out var neighbor)) continue;
166
167                 if (neighbor.Region == currentRegion) continue;
168
169                 currentCell.SetIsEdge(true);
170
171                 break;
172             }
173         }
174     }
175 }
176
177 private static void FindDistancesToEdges(Grid<VoronoiFieldCell> grid)
178 {
179     var flowFieldGrid = new Grid<FlowFieldCell>(grid.Size, grid.Center,
180     ↪ grid.CellSize, grid.UnwalkableMask,
181     ↪ (center, _, x, y) => new FlowFieldCell(center, true, x, y),
182     ↪ grid.Obstacles);
183
184     var startCells = new List<FlowFieldCell>();
185
186     for (var x = 0; x < grid.Width; x++)
187     {
188         for (var y = 0; y < grid.Height; y++)
189         {
190             var voronoiFieldCell = grid.Cells[x, y];
191
192             if (!voronoiFieldCell.IsEdge) continue;
193
194             var flowFieldCell = flowFieldGrid.Cells[x, y];
195
196             startCells.Add(flowFieldCell);
197         }
198     }
199
200     FlowField.CalculateFlowField(flowFieldGrid, startCells);

```

```
200     for (var x = 0; x < grid.Width; x++)
201     {
202         for (var y = 0; y < grid.Height; y++)
203         {
204             var voronoiFieldCell = grid.Cells[x, y];
205             var flowFieldCell = flowFieldGrid.Cells[x, y];
206             var closestEdgeDistance = flowFieldCell.Cost;
207             voronoiFieldCell.SetClosestEdgeDistance(closestEdgeDistance);
208         }
209     }
210 }
211 }
212 }
213 }
214 }
```

# ANEXO L – Classe ReedsShepp

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using UnityEngine;
5  using Path = System.Collections.Generic.List<Movement>;
6
7  public class ReedsShepp
8  {
9      private static float PathLength(Path path)
10     {
11         return path.Sum(movement => movement.Distance);
12     }
13
14     public static Path GetOptimalPath(float x1, float y1, float theta1, float x2,
15     ↪ float y2, float theta2)
16     {
17         var paths = GetAllPaths(x1, y1, theta1, x2, y2, theta2);
18         var optimalPath = paths.OrderBy(PathLength).First();
19         return optimalPath;
20     }
21
22     public static List<Path> GetAllPaths(float x1, float y1, float theta1, float x2,
23     ↪ float y2, float theta2)
24     {
25         var pathFuncs = new List<Func<float, float, float, Path>>
26         { Path1, Path2, Path3, Path4, Path5, Path6, Path7, Path8, Path9, Path10,
27         ↪ Path11, Path12 };
28
29         var paths = new List<Path>();
30
31         var (x, y, theta) = ChangeOfBasis(
32             x1,
33             y1,
34             90f - theta1,
35             x2,
36             y2,
37             90f - theta2,
38             Constants.CarTurningRadius
39         );
40
41         foreach (var getPath in pathFuncs)
42         {
43             paths.Add(getPath(x, y, theta));
44             paths.Add(TimeFlip(getPath(-x, y, -theta)));
45             paths.Add(Reflect(getPath(x, -y, -theta)));
46             paths.Add(Reflect(TimeFlip(getPath(-x, -y, theta))));
47         }
48
49         for (var i = paths.Count - 1; i >= 0; i--)
50         {
51             if (paths[i].Any(movement => float.IsNaN(movement.Distance)))
52             {
53                 paths.RemoveAt(i);
54                 continue;
55             }
56
57             paths[i] = paths[i]
58                 .Where(movement => movement.Distance != 0)
59                 .Select(movement =>
60                 {
61                     movement.Distance *= Constants.CarTurningRadius;
62                     return movement;
63                 })

```

```

63         .ToList();
64     }
65
66     paths.RemoveAll(path => path.Count == 0);
67
68     return paths;
69 }
70
71 private static Path TimeFlip(Path path)
72 {
73     foreach (var movement in path)
74     {
75         movement.ReverseGear();
76     }
77
78     return path;
79 }
80
81 private static Path Reflect(Path path)
82 {
83     foreach (var movement in path)
84     {
85         movement.ReverseSteering();
86     }
87
88     return path;
89 }
90
91 private static (float x, float y, float theta) ChangeOfBasis(float x1, float y1,
92 ↪ float theta1, float x2, float y2,
93     float theta2, float turningRadius = 1f)
94 {
95     var dx = (x2 - x1) / turningRadius;
96     var dy = (y2 - y1) / turningRadius;
97
98     var theta = theta2 - theta1;
99
100     theta1 *= Mathf.Deg2Rad;
101
102     var x = dx * Mathf.Cos(theta1) + dy * Mathf.Sin(theta1);
103     var y = -dx * Mathf.Sin(theta1) + dy * Mathf.Cos(theta1);
104
105     return (x, y, theta);
106 }
107
108 private static Path Path1(float x, float y, float phi)
109 {
110     phi *= Mathf.Deg2Rad;
111
112     var (u, t) = R(x - Mathf.Sin(phi), y - 1f + Mathf.Cos(phi));
113     var v = M(phi - t);
114
115     var path = new Path
116     {
117         new(t, Movement.Steering.Left, Movement.Gear.Forward),
118         new(u, Movement.Steering.Straight, Movement.Gear.Forward),
119         new(v, Movement.Steering.Left, Movement.Gear.Forward)
120     };
121
122     return path;
123 }
124
125 private static Path Path2(float x, float y, float phi)
126 {
127     phi = M(phi * Mathf.Deg2Rad);
128
129     var (rho, t1) = R(x + Mathf.Sin(phi), y - 1f - Mathf.Cos(phi));
130
131     Path path = new();
132
133     if (rho * rho >= 4f)
134     {

```

```

134         var u = Mathf.Sqrt(rho * rho - 4f);
135         var t = M(t1 + Mathf.Atan2(2f, u));
136         var v = M(t - phi);
137
138         path = new Path
139         {
140             new(t, Movement.Steering.Left, Movement.Gear.Forward),
141             new(u, Movement.Steering.Straight, Movement.Gear.Forward),
142             new(v, Movement.Steering.Right, Movement.Gear.Forward)
143         };
144     }
145
146     return path;
147 }
148
149 private static Path Path3(float x, float y, float phi)
150 {
151     phi *= Mathf.Deg2Rad;
152
153     var xi = x - Mathf.Sin(phi);
154     var eta = y - 1f + Mathf.Cos(phi);
155
156     var (rho, theta) = R(xi, eta);
157
158     Path path = new();
159
160     if (rho <= 4f)
161     {
162         var A = Mathf.Acos(rho / 4f);
163         var t = M(theta + Mathf.PI / 2f + A);
164         var u = M(Mathf.PI - 2f * A);
165         var v = M(phi - t - u);
166
167         path = new Path
168         {
169             new(t, Movement.Steering.Left, Movement.Gear.Forward),
170             new(u, Movement.Steering.Right, Movement.Gear.Backward),
171             new(v, Movement.Steering.Left, Movement.Gear.Forward)
172         };
173     }
174
175     return path;
176 }
177
178 private static Path Path4(float x, float y, float phi)
179 {
180     phi *= Mathf.Deg2Rad;
181
182     var xi = x - Mathf.Sin(phi);
183     var eta = y - 1f + Mathf.Cos(phi);
184
185     var (rho, theta) = R(xi, eta);
186
187     Path path = new();
188
189     if (rho <= 4f)
190     {
191         var A = Mathf.Acos(rho / 4f);
192         var t = M(theta + Mathf.PI / 2f + A);
193         var u = M(Mathf.PI - 2f * A);
194         var v = M(t + u - phi);
195
196         path = new Path
197         {
198             new(t, Movement.Steering.Left, Movement.Gear.Forward),
199             new(u, Movement.Steering.Right, Movement.Gear.Backward),
200             new(v, Movement.Steering.Left, Movement.Gear.Backward)
201         };
202     }
203
204     return path;
205 }

```

```

206
207 private static Path Path5(float x, float y, float phi)
208 {
209     phi *= Mathf.Deg2Rad;
210
211     var xi = x - Mathf.Sin(phi);
212     var eta = y - 1f + Mathf.Cos(phi);
213
214     var (rho, theta) = R(xi, eta);
215
216     Path path = new();
217
218     if (rho <= 4f)
219     {
220         var u = Mathf.Acos(1f - rho * rho / 8f);
221         var A = Mathf.Asin(2f * Mathf.Sin(u) / rho);
222         var t = M(theta + Mathf.PI / 2f - A);
223         var v = M(t - u - phi);
224
225         path = new Path
226         {
227             new(t, Movement.Steering.Left, Movement.Gear.Forward),
228             new(u, Movement.Steering.Right, Movement.Gear.Forward),
229             new(v, Movement.Steering.Left, Movement.Gear.Backward)
230         };
231     }
232
233     return path;
234 }
235
236 private static Path Path6(float x, float y, float phi)
237 {
238     phi *= Mathf.Deg2Rad;
239
240     var xi = x + Mathf.Sin(phi);
241     var eta = y - 1f - Mathf.Cos(phi);
242
243     var (rho, theta) = R(xi, eta);
244
245     float t, u, v, A;
246
247     Path path = new();
248
249     if (rho <= 4f)
250     {
251         if (rho <= 2f)
252         {
253             A = Mathf.Acos((rho + 2f) / 4f);
254             t = M(theta + Mathf.PI / 2f + A);
255             u = M(A);
256             v = M(phi - t + 2f * u);
257         }
258         else
259         {
260             A = Mathf.Acos((rho - 2f) / 4f);
261             t = M(theta + Mathf.PI / 2f - A);
262             u = M(Mathf.PI - A);
263             v = M(phi - t + 2f * u);
264         }
265
266         path = new Path
267         {
268             new(t, Movement.Steering.Left, Movement.Gear.Forward),
269             new(u, Movement.Steering.Right, Movement.Gear.Forward),
270             new(u, Movement.Steering.Left, Movement.Gear.Backward),
271             new(v, Movement.Steering.Right, Movement.Gear.Backward)
272         };
273     }
274
275     return path;
276 }
277

```

```

278 private static Path Path7(float x, float y, float phi)
279 {
280     phi *= Mathf.Deg2Rad;
281
282     var xi = x + Mathf.Sin(phi);
283     var eta = y - 1f - Mathf.Cos(phi);
284
285     var (rho, theta) = R(xi, eta);
286     var u1 = (20f - rho * rho) / 16f;
287
288     Path path = new();
289
290     if (rho <= 6f && u1 >= 0 && u1 <= 1f)
291     {
292         var u = Mathf.Acos(u1);
293         var A = Mathf.Asin(2f * Mathf.Sin(u) / rho);
294         var t = M(theta + Mathf.PI / 2f + A);
295         var v = M(t - phi);
296
297         path = new Path
298         {
299             new(t, Movement.Steering.Left, Movement.Gear.Forward),
300             new(u, Movement.Steering.Right, Movement.Gear.Backward),
301             new(u, Movement.Steering.Left, Movement.Gear.Backward),
302             new(v, Movement.Steering.Right, Movement.Gear.Forward)
303         };
304     }
305
306     return path;
307 }
308
309 private static Path Path8(float x, float y, float phi)
310 {
311     phi *= Mathf.Deg2Rad;
312
313     var xi = x - Mathf.Sin(phi);
314     var eta = y - 1f + Mathf.Cos(phi);
315
316     var (rho, theta) = R(xi, eta);
317
318     Path path = new();
319
320     if (rho >= 2f)
321     {
322         var u = Mathf.Sqrt(rho * rho - 4f) - 2f;
323         var A = Mathf.Atan2(2f, u + 2f);
324         var t = M(theta + Mathf.PI / 2f + A);
325         var v = M(t - phi + Mathf.PI / 2f);
326
327         path = new Path
328         {
329             new(t, Movement.Steering.Left, Movement.Gear.Forward),
330             new(Mathf.PI / 2f, Movement.Steering.Right, Movement.Gear.Backward),
331             new(u, Movement.Steering.Straight, Movement.Gear.Backward),
332             new(v, Movement.Steering.Left, Movement.Gear.Backward)
333         };
334     }
335
336     return path;
337 }
338
339 private static Path Path9(float x, float y, float phi)
340 {
341     phi *= Mathf.Deg2Rad;
342
343     var xi = x - Mathf.Sin(phi);
344     var eta = y - 1f + Mathf.Cos(phi);
345
346     var (rho, theta) = R(xi, eta);
347
348     Path path = new();
349

```

```

350     if (rho >= 2f)
351     {
352         var u = Mathf.Sqrt(rho * rho - 4f) - 2f;
353         var A = Mathf.Atan2(u + 2f, 2f);
354         var t = M(theta + Mathf.PI / 2f - A);
355         var v = M(t - phi - Mathf.PI / 2f);
356
357         path = new Path
358         {
359             new(t, Movement.Steering.Left, Movement.Gear.Forward),
360             new(u, Movement.Steering.Straight, Movement.Gear.Forward),
361             new(Mathf.PI / 2f, Movement.Steering.Right, Movement.Gear.Forward),
362             new(v, Movement.Steering.Left, Movement.Gear.Backward)
363         };
364     }
365
366     return path;
367 }
368
369 private static Path Path10(float x, float y, float phi)
370 {
371     phi *= Mathf.Deg2Rad;
372
373     var xi = x + Mathf.Sin(phi);
374     var eta = y - 1f - Mathf.Cos(phi);
375
376     var (rho, theta) = R(xi, eta);
377
378     Path path = new();
379
380     if (rho >= 2f)
381     {
382         var t = M(theta + Mathf.PI / 2f);
383         var u = rho - 2f;
384         var v = M(phi - t - Mathf.PI / 2f);
385
386         path = new Path
387         {
388             new(t, Movement.Steering.Left, Movement.Gear.Forward),
389             new(Mathf.PI / 2f, Movement.Steering.Right, Movement.Gear.Backward),
390             new(u, Movement.Steering.Straight, Movement.Gear.Backward),
391             new(v, Movement.Steering.Right, Movement.Gear.Backward)
392         };
393     }
394
395     return path;
396 }
397
398 private static Path Path11(float x, float y, float phi)
399 {
400     phi *= Mathf.Deg2Rad;
401
402     var xi = x + Mathf.Sin(phi);
403     var eta = y - 1f - Mathf.Cos(phi);
404
405     var (rho, theta) = R(xi, eta);
406
407     Path path = new();
408
409     if (rho >= 2f)
410     {
411         var t = M(theta);
412         var u = rho - 2f;
413         var v = M(phi - t - Mathf.PI / 2f);
414
415         path = new Path
416         {
417             new(t, Movement.Steering.Left, Movement.Gear.Forward),
418             new(u, Movement.Steering.Straight, Movement.Gear.Forward),
419             new(Mathf.PI / 2f, Movement.Steering.Left, Movement.Gear.Forward),
420             new(v, Movement.Steering.Right, Movement.Gear.Backward)
421         };

```

```

422     }
423
424     return path;
425 }
426
427 private static Path Path12(float x, float y, float phi)
428 {
429     phi *= Mathf.Deg2Rad;
430
431     var xi = x + Mathf.Sin(phi);
432     var eta = y - 1f - Mathf.Cos(phi);
433
434     var (rho, theta) = R(xi, eta);
435
436     Path path = new();
437
438     if (rho >= 4f)
439     {
440         var u = Mathf.Sqrt(rho * rho - 4f) - 4f;
441         var A = Mathf.Atan2(2f, u + 4f);
442         var t = M(theta + Mathf.PI / 2f + A);
443         var v = M(t - phi);
444
445         path = new Path
446         {
447             new(t, Movement.Steering.Left, Movement.Gear.Forward),
448             new(Mathf.PI / 2f, Movement.Steering.Right, Movement.Gear.Backward),
449             new(u, Movement.Steering.Straight, Movement.Gear.Backward),
450             new(Mathf.PI / 2f, Movement.Steering.Left, Movement.Gear.Backward),
451             new(v, Movement.Steering.Right, Movement.Gear.Forward)
452         };
453     }
454
455     return path;
456 }
457
458 private static float M(float theta)
459 {
460     theta %= (2f * Mathf.PI);
461     if (theta < -Mathf.PI) return theta + 2f * Mathf.PI;
462     if (theta >= Mathf.PI) return theta - 2f * Mathf.PI;
463     return theta;
464 }
465
466 private static (float, float) R(float x, float y)
467 {
468     var r = Mathf.Sqrt(x * x + y * y);
469     var theta = Mathf.Atan2(y, x);
470     return (r, theta);
471 }
472 }
```

## ANEXO M – Classe Node

```

1  using System;
2  using UnityEngine;
3
4  public class Node : IComparable<Node>
5  {
6      public Vector3 RearWheelPosition { get; set; }
7
8      public float HeadingAngle { get; set; }
9
10     public Movement MovementInfo { get; set; }
11
12     public Node Parent { get; set; }
13
14     public float ChangingDirectionCost =>
15         Parent != null && Parent.MovementInfo.GearVal != MovementInfo.GearVal ?
16         ↪ Constants.ChangingDirectionCost : 0;
17
18     public float IsReversingCost => MovementInfo.GearVal == Movement.Gear.Backward ?
19     ↪ Constants.ReversingCost : 0;
20
21     public float IsTurningCost => MovementInfo.SteeringVal !=
22     ↪ Movement.Steering.Straight ? Constants.TurningCost : 0;
23
24     public float ExpansionCost => MovementInfo.Distance + ChangingDirectionCost +
25     ↪ IsReversingCost + IsTurningCost;
26
27     public float gCost => Parent != null ? Parent.gCost + ExpansionCost : 0;
28
29     public float hCost { get; set; }
30
31     public float fCost => gCost + hCost;
32
33     public Node(Vector3 rearWheelPosition, float headingAngle, Movement movement, Node
34     ↪ parent)
35     {
36         RearWheelPosition = rearWheelPosition;
37         HeadingAngle = headingAngle;
38         MovementInfo = movement;
39         Parent = parent;
40     }
41
42     public int CompareTo(Node other)
43     {
44         var compare = fCost.CompareTo(other.fCost);
45
46         if (compare == 0)
47         {
48             compare = hCost.CompareTo(other.hCost);
49         }
50
51         return -compare;
52     }
53 }

```

## ANEXO N – Classe HybridAStar

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using UnityEngine;
5  using Path = System.Collections.Generic.List<Movement>;
6
7  public class HybridAStar
8  {
9      public static Path FindPath(Grid<Cell> grid, Car startCar, Car goalCar, float[,]  

10         ↪ euclideanCosts,  

11         float[,]  

12         ↪ holonomicCosts, float[,]  

13         ↪ flowFieldCosts, float[,]  

14         ↪ voronoiFieldCosts,  

15         ↪ out List<Node> finalNodes,  

16         ↪ out List<Node> expandedNodes, bool analyticalExpansion = true)
17      {
18          // TODO: Validate start and goal positions
19
20          finalNodes = new List<Node>();
21          expandedNodes = new List<Node>();
22
23          var openSet = new PriorityQueue<Node>(Constants.Iterations * 6);
24          var closedSetList = new HashSet<int>[grid.Width, grid.Height];
25          var lowestCostSetList = new Dictionary<int, Node>[grid.Width, grid.Height];
26
27          for (var i = 0; i < grid.Width; i++)
28          {
29              for (var j = 0; j < grid.Height; j++)
30              {
31                  closedSetList[i, j] = new HashSet<int>();
32                  lowestCostSetList[i, j] = new Dictionary<int, Node>();
33              }
34          }
35
36          var startMovement = new Movement(0, Movement.Steering.Straight,  

37             ↪ Movement.Gear.Forward);
38
39          var startNode = new Node(startCar.RearWheelPosition, startCar.HeadingAngle,  

40             ↪ startMovement, parent: null);
41
42          openSet.Enqueue(startNode);
43
44          Path reedsSheppPath = null;
45
46          Node finalNode = null;
47
48          var iterations = 0;
49
50          while (iterations < Constants.Iterations)
51          {
52              iterations++;
53
54              var currentNode = openSet.Dequeue();
55
56              if (!grid.TryGetCellFromWorldPosition(currentNode.RearWheelPosition, out  

57                 ↪ var currentCell))
58              {
59                  throw new Exception("The current node is outside the grid.");
60              }
61
62              // TODO: Run the analytical expansion only when the goal is close to the  

63                 ↪ current node
64              if (analyticalExpansion && TryAnalyticalExpansion(grid,  

65                 ↪ currentNode.RearWheelPosition,

```

```

59         currentNode.HeadingAngle, goalCar.RearWheelPosition,
60         ↪ goalCar.HeadingAngle,
61         ↪ out reedsSheppPath))
62     {
63         finalNode = currentNode;
64         break;
65     }
66     var distance = (currentNode.RearWheelPosition -
67     ↪ goalCar.RearWheelPosition).magnitude;
68     var goalPositionReached = distance < Constants.GoalDistanceThreshold;
69     var goalAngleReached = Math.Abs(currentNode.HeadingAngle -
70     ↪ goalCar.HeadingAngle) <
71     ↪ Constants.GoalHeadingAngleThreshold;
72     var goalReached = goalPositionReached && goalAngleReached;
73     if (goalReached)
74     {
75         finalNode = currentNode;
76         break;
77     }
78     var roundedHeading =
79     ↪ Utils.RoundValueToNearestStep(currentNode.HeadingAngle,
80     ↪ Constants.HeadingAngleStep);
81     var closedSet = closedSetList[currentCell.GridPosition.x,
82     ↪ currentCell.GridPosition.y];
83     if (!closedSet.Add(roundedHeading)) continue;
84     expandedNodes.Add(currentNode);
85     foreach (var child in GetNodeChildren(grid, currentNode))
86     {
87         if (!grid.TryGetCellFromWorldPosition(child.RearWheelPosition, out var
88         ↪ childCell)) continue;
89         var x = childCell.GridPosition.x;
90         var y = childCell.GridPosition.y;
91         child.hCost += Mathf.Max(euclideanCosts[x, y], holonomicCosts[x, y]);
92         child.hCost += flowFieldCosts[x, y] * voronoiFieldCosts[x, y] * 0.35f;
93         child.hCost += flowFieldCosts[x, y] * 1.1f;
94         closedSet = closedSetList[x, y];
95         var roundedChildHeadingAngle =
96         ↪ Utils.RoundValueToNearestStep(child.HeadingAngle,
97         ↪ Constants.HeadingAngleStep);
98         if (closedSet.Contains(roundedChildHeadingAngle)) continue;
99         var lowestCostSet = lowestCostSetList[x, y];
100         if (lowestCostSet.TryGetValue(roundedChildHeadingAngle, out var
101         ↪ lowestCostNode))
102         {
103             if (child.gCost > lowestCostNode.gCost) continue;
104             lowestCostSet.Remove(roundedChildHeadingAngle);
105             lowestCostSet.Add(roundedChildHeadingAngle, child);
106         }
107         else
108         {
109             lowestCostSet.Add(roundedChildHeadingAngle, child);
110         }
111     }
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 }
120 }

```

```

121         openSet.Enqueue(child);
122     }
123 }
124
125 if (finalNode == null) return null;
126
127 var path = RetracePath(finalNode, finalNodes);
128
129 if (reedsSheppPath != null)
130 {
131     path.AddRange(reedsSheppPath);
132 }
133
134 return path;
135 }
136
137 private static List<Node> GetNodeChildren(Grid<Cell> grid, Node node)
138 {
139     return ((Movement.Gear[])Enum.GetValues(typeof(Movement.Gear)))
140         .SelectMany(gear =>
141             ((Movement.Steering[])Enum.GetValues(typeof(Movement.Steering)))
142                 .Select(steering =>
143                     {
144                         var movement = new Movement(Constants.DriveDistance, steering,
145                             gear);
146
147                         var (childRearWheelPosition, childHeadingAngle) =
148                             CarModel.CalculateMovement(node.RearWheelPosition,
149                                 node.HeadingAngle, movement);
150
151                         if (!grid.IsPositionWalkable(childRearWheelPosition))
152                         {
153                             return null;
154                         }
155
156                         var child = new Node(childRearWheelPosition, childHeadingAngle,
157                             movement, node);
158
159                         return child;
160                     })
161                 .Where(child => child != null)
162                 .ToList();
163 }
164
165 private static bool TryAnalyticalExpansion(Grid<Cell> grid, Vector3
166     currentPosition, float currentHeadingAngle,
167     Vector3 goalPosition, float goalHeadingAngle, out Path reedsSheppPath)
168 {
169     reedsSheppPath = ReedsShepp.GetOptimalPath(currentPosition.x,
170         currentPosition.z, currentHeadingAngle,
171         goalPosition.x, goalPosition.z, goalHeadingAngle);
172
173     if (reedsSheppPath == null)
174         return false;
175
176     var (waypoints, angles) =
177         PathBuilder.GenerateWaypoints(currentPosition, currentHeadingAngle,
178             reedsSheppPath, 0.35f);
179
180     const float halfLength = Constants.CarLength / 2;
181     const float halfWidth = Constants.CarWidth / 2;
182     const float heightOffset = 0.1f;
183
184     for (var i = 0; i < waypoints.Count; i++)
185     {
186         var waypoint = waypoints[i];
187         var angle = angles[i];
188         var rotation = Quaternion.Euler(0, angle, 0);
189
190         var centerOffset = rotation * new Vector3(0, 0, Constants.CarLength / 2 -
191             Constants.CarRearOverhang);

```

```

185         var vehicleCenter = waypoint + centerOffset;
186
187         Vector3[] corners =
188         {
189             vehicleCenter + rotation * new Vector3(-halfWidth, 0, halfLength),
190             vehicleCenter + rotation * new Vector3(halfWidth, 0, halfLength),
191             vehicleCenter + rotation * new Vector3(halfWidth, 0, -halfLength),
192             vehicleCenter + rotation * new Vector3(-halfWidth, 0, -halfLength)
193         };
194
195         for (var j = 0; j < corners.Length; j++)
196         {
197             var start = corners[j] + Vector3.up * heightOffset;
198             var end = corners[(j + 1) % corners.Length] + Vector3.up *
199                 ↪ heightOffset;
200
201             if (Physics.Linecast(start, end, grid.UnwalkableMask) ||
202                 Physics.CheckSphere(start, 0.1f, grid.UnwalkableMask))
203             {
204                 reedsSheppPath = null;
205                 return false;
206             }
207         }
208
209         return true;
210     }
211
212     private static Path RetracePath(Node node, List<Node> finalNodes)
213     {
214         Path path = new();
215
216         var currentNode = node;
217
218         while (currentNode != null)
219         {
220             path.Add(currentNode.MovementInfo);
221
222             finalNodes.Add(currentNode);
223
224             currentNode = currentNode.Parent;
225         }
226
227         path.Reverse();
228
229         finalNodes.Reverse();
230
231         return path;
232     }
233 }

```

## ANEXO O – Classe PathBuilder

```

1  using System.Collections.Generic;
2  using UnityEngine;
3  using Path = System.Collections.Generic.List<Movement>;
4
5  public static class PathBuilder
6  {
7      public static (List<Vector3>, List<float>) GenerateWaypoints(Vector3
      ↪   currentPosition, float currentHeading,
8          Path path,
9          float stepDistance)
10     {
11         var waypoints = new List<Vector3>()
12         {
13             currentPosition
14         };
15
16         var headings = new List<float>()
17         {
18             currentHeading
19         };
20
21         foreach (var movement in path)
22         {
23             var remainingDistance = movement.Distance;
24
25             while (remainingDistance > 0)
26             {
27                 var stepSize = Mathf.Min(stepDistance, remainingDistance);
28                 var stepMovement = new Movement(stepSize, movement.SteeringVal,
29                 ↪   movement.GearVal);
30
31                 (currentPosition, currentHeading) =
32                     CarModel.CalculateMovement(currentPosition, currentHeading,
33                     ↪   stepMovement);
34
35                 waypoints.Add(currentPosition);
36                 headings.Add(currentHeading);
37
38                 remainingDistance -= stepSize;
39             }
40         }
41         return (waypoints, headings);
42     }

```