Universidade Federal
de São João del-Rei

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Bárbara Pereira Medeiros Dias
Gabriel de Paula Meira
Pedro Antônio Machado Costa

River Raid: Desenvolvimento de um Agente Automatizado para jogar

São João del Rei
2025

Bárbara Pereira Medeiros Dias
Gabriel de Paula Meira
Pedro Antônio Machado Costa

River Raid: Desenvolvimento de um Agente Automatizado para jogar

Trabalho apresentado ao curso de Ciência da Computação da Universidade Federal de São João del Rei como requisito parcial para avaliação do componente curricular de Robótica Computacional.
Orientador: Prof. Dr. Marcos Antonio de Matos Laia

São João del Rei
2025

SUMÁRIO

1 INTRODUÇÃO

Na disciplina de Robótica Computacional, foi proposto o desenvolvimento de agentes capazes de explorar e interagir em ambientes virtuais, utilizando conceitos e técnicas aplicadas em robótica autônoma. Para isso, implementamos agentes controladores no jogo River Raid (Atari 2600), um jogo clássico de exploração em 2D que simula um ambiente de navegação com obstáculos e inimigos, sendo um cenário ideal para treinar algoritmos de navegação autônoma.

O objetivo do trabalho era projetar e implementar estratégias que permitissem ao agente avançar no cenário do jogo de forma eficiente, utilizando técnicas de campo potencial, previsão de movimento com filtros de Kalman e aprendizado por reforço (Q-Learning), alinhadas com os tópicos trabalhados ao longo da disciplina. Essas técnicas simulam a percepção, o planejamento de trajetória e a tomada de decisão de robôs autônomos em ambientes dinâmicos.

A implementação foi realizada utilizando a API OpenAI Gym Retro integrada ao Python, com a construção de pipelines de treinamento e avaliação do agente em fases progressivas, buscando maximizar o avanço no cenário enquanto evita obstáculos e gerencia recursos (como combustível e munição). Nosso agente foi testado para superar ao menos uma fase do River Raid de forma autônoma, utilizando aprendizado baseado em interação com o ambiente para aprimorar sua performance ao longo dos episódios.

Este trabalho proporcionou um ambiente prático para aplicar conceitos de robótica, reforçando a compreensão de algoritmos de navegação e aprendizado em agentes autônomos, além de promover habilidades de programação e análise de desempenho em sistemas de controle inteligente.

2 METODOLOGIA

Para a realização deste trabalho na disciplina de Robótica Computacional, o desenvolvimento foi organizado em três etapas progressivas, cada uma utilizando uma técnica distinta de controle e navegação em ambientes dinâmicos, aplicadas ao jogo *River Raid*. Essa divisão permitiu compreender gradualmente a evolução de estratégias de controle autônomo, passando de métodos reativos a métodos de previsão e, por fim, ao aprendizado por reforço, aproximando o estudo teórico de sua aplicação prática no controle de agentes em ambientes reais.

2.1 SOBRE O JOGO

O *River Raid* foi desenvolvido por Carol Shaw e lançado pela Activision em 1982 para o Atari 2600. O jogo consiste em pilotar um avião por um rio repleto de obstáculos, pontes, inimigos móveis e zonas de combustível, exigindo controle preciso e estratégias de navegação para avançar pelas fases e acumular pontuação.

Na Figura 1 são apresentados a capa oficial do jogo (Figura 1a) e uma captura de tela representativa do ambiente de jogo (Figura 1b), demonstrando o cenário vertical e os elementos que compõem a dinâmica do River Raid.

(a) *Capa* ***(b)*** *Captura de tela*

Figura 1: *River Raid © Activision*

O jogo é contínuo, havendo pontuações atribuídas aos feitos até que as vidas acabem; quanto mais tempo durar a jogatina, melhor é o resultado obtido. Conforme as fases vão progredindo, o nível de dificuldade aumenta e o espaço para realizar as manobras diminui [1].

A Figura 2 ilustra a barra de status do jogo, onde é possível visualizar informações essenciais como a pontuação atual do jogador, o nível de combustível restante e o número de vidas disponíveis, sendo variáveis importantes para a tomada de decisão durante a navegação.

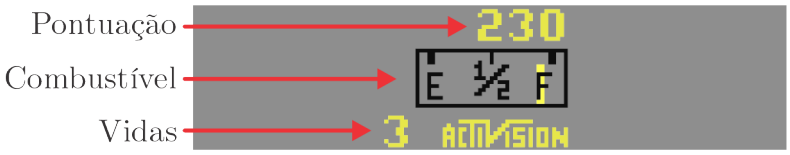


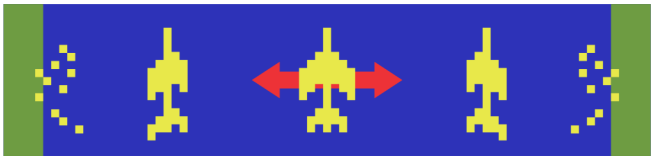
Figura 2: *Barra de status do jogo*

2.1.1 Cenário e Ações

O cenário do *River Raid* se desloca verticalmente, enquanto o jogador controla a nave lateralmente, podendo acelerar, freare disparar para destruir pontes e inimigos, sendo essas ações essenciais para a progressão no jogo.

A Figura 3 ilustra a movimentação do jogador, destacando as possibilidades de deslocamento lateral para desviar de obstáculos, alinhar-se a zonas seguras e posicionar-se para disparos. Essas movimentações são fundamentais para o agente otimizar trajetórias, evitar colisões e coletar combustível de forma eficiente durante a execução autônoma do jogo.

Figura 3: *Movimentação do jogador*



A área navegável é representada pela cor azul da água do cenário. O avião do jogador não pode, em nenhum momento, colidir com inimigos ou com as paredes do cenário, resultando em morte (perda de uma vida).

A Figura 4 apresenta uma ponte, elemento presente ao longo do percurso e que precisa ser destruído para liberar a passagem. Cada ponte define o início da próxima fase. Caso o jogador colida com algum obstáculo, o ponto de ressurgimento será a última ponte destruída.

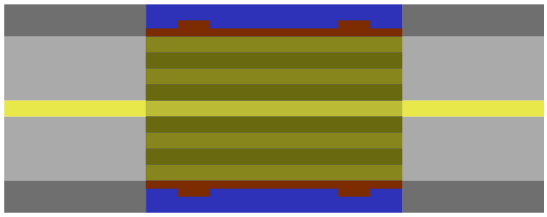


Figura 4: *Ponte*

2.1.2 Inimigos

O jogo possui diversos tipos de inimigos, incluindo barcos, helicópteros e aviões, que cruzam a tela de maneira imprevisível, além de obstáculos fixos e pontes. A identificação e a evasão destes elementos são essenciais para a sobrevivência e progresso do jogador.

Na Figura 5 são apresentados os *sprites* dos inimigos existentes, destacando visualmente as principais ameaças que o agente precisa identificar e evitar durante a navegação. Esses elementos foram utilizados no processo de segmentação de imagens, permitindo ao agente detectar a presença dos inimigos em tempo real e ajustar sua trajetória para desvio ou eliminação de alvos, dependendo da estratégia empregada em cada técnica implementada.

Figura 5: *Sprites dos inimigos existentes*

2.1.3 Combustível

O combustível do avião é consumido constantemente durante o jogo, sendo necessário passar por tanques espalhados ao longo do trajeto para evitar a queda. O gerenciamento do combustível é uma variável crítica de sobrevivência do agente, visto que seu esgotamento resulta na perda imediata de uma vida.

Na Figura 6 é apresentado o *sprite* do combustível, identificado pela inscrição `FUEL`. Este elemento foi utilizado nos processos de segmentação de cor e detecção em tempo real, permitindo ao agente priorizar trajetórias que incluam a coleta de combustível sempre que o nível estiver crítico, garantindo maior longevidade durante a navegação.

Combustível

Figura 6: *Sprite do Combustível*

2.2 MOVIMENTAÇÃO COM CAMPO POTENCIAL

A primeira técnica implementada foi a movimentação com campo potencial, técnica baseada na geração de campos de forças atrativas (zonas seguras e combustível) e repulsivas (obstáculos e inimigos). Essa técnica permite ao agente reagir dinamicamente ao cenário sem conhecimento prévio do mapa, ajustando sua trajetória em tempo real de forma reativa.

2.2.1 Identificação dos objetos

Para identificar os elementos relevantes do cenário, foi utilizada segmentação de cores no espaço **HSV** (Hue, Saturation, Value) [2], permitindo distinguir de forma robusta os objetos independentemente de nuances de cor. Foram definidas máscaras para cada objeto de interesse: combustível, obstáculos, pontes e inimigos.

A Figura 7 (Figura 7) ilustra os intervalos de cores utilizados para segmentação, fundamentais para o pipeline de percepção do agente.

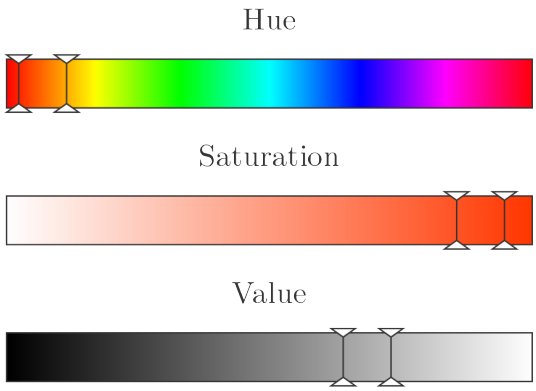


Figura 7: *Representação de intervalos de cores HSV utilizados para segmentação*

Com as máscaras, foi possível realizar a extração das coordenadas dos elementos relevantes para o cálculo do campo potencial e realizar a tomada de decisão do agente.

2.2.2 Tomada de decisão

A lógica de decisão, representada pelo Algoritmo 1, é simples e eficiente para a maioria das ocasiões. Contudo, em situações específicas, decisões podem conflitar devido ao fato de o agente não ser capaz de lidar com ambientes muito dinâmicos. Há mais ações executadas, o processo foi simplificado para facilitar o entendimento.

Algorithm 1 Lógica de decisão para Campo Potencial

```
if enemy_will_crash then
    move_to_element(enemy_will_crash, avoid=True)
else if should_center_passing then
    move_to_element(should_center_passing)
else if aiming_enemy then
    move_to_element(aiming_enemy)
else if almost_aiming_enemy then
    move_to_element(almost_aiming_enemy)
else if next_fuel then
    move_to_element(next_fuel)
```

2.3 FILTRO DE KALMAN

O Filtro de Kalman é um algoritmo utilizado para estimar o estado de um sistema dinâmico a partir de uma série de medições incompletas ou ruidosas. Em vez de utilizar apenas uma única medição, ele combina um modelo de como o sistema se comporta com as medições reais ao longo do tempo para produzir uma estimativa muito mais precisa e estável [3].

No contexto do *River Raid*, o problema principal que motivou a implementação deste filtro é o momento que surge um avião inimigo que entra pela lateral e se move em uma trajetória diagonal. Apenas observar sua posição atual (x, y) em cada quadro não revela sua velocidade ou sua intenção de trajetória futura.

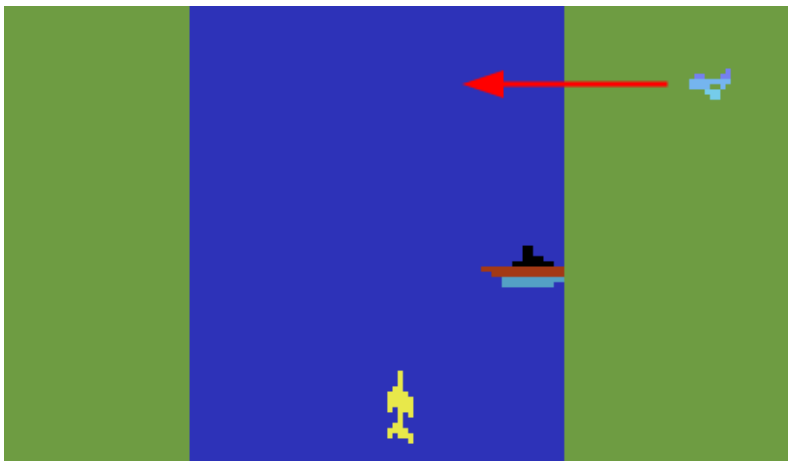


Figura 8: *Inimigo vindo da lateral, sem respeitar as paredes*

O Filtro de Kalman é a solução ideal para este problema, pois ele pode usar a sequência de posições observadas para estimar a velocidade (v_x, v_y) do avião e, com isso, prever sua trajetória futura com alta precisão. Isso permite que o *Bot* deixe de ser puramente reativo e passe a agir de forma preditiva.

2.3.1 Funcionamento do Filtro de Kalman

O funcionamento do Filtro de Kalman baseia-se em um ciclo contínuo de duas etapas: predição e correção:

- **Predição**: Nesta fase, o filtro usa seu modelo interno do sistema (por exemplo, "um objeto em movimento tende a manter sua velocidade") para prever qual será o próximo estado do sistema. No caso do avião, ele prevê onde o avião estará no próximo instante de tempo.
- **Correção**: Em seguida, o filtro recebe uma medição real do sistema (a posição (x, y) do avião detectada na tela). Ele compara essa medição com sua predição. A diferença entre os dois é usada para corrigir a estimativa do estado. Se a medição for muito diferente da predição, o filtro ajusta sua estimativa para se aproximar mais da medição real.

Ao repetir esse ciclo, o filtro "aprende"o estado verdadeiro do sistema, incluindo variáveis que não são medidas diretamente, como a velocidade. Ele filtra o ruído (pequenas imprecisões na detecção) e fornece uma estimativa suave e confiável da trajetória do objeto.

2.3.2 Simulação e Validação do Conceito

Antes de integrar o filtro diretamente ao Bot, foi desenvolvida uma simulação em Python para validar sua eficácia em nosso cenário específico. O objetivo era confirmar que o filtro conseguiria, a partir de medições de posição com ruído, estimar com precisão a trajetória de um objeto em movimento diagonal e prever seu ponto de interceptação com o eixo y=0.

O script simula os seguintes elementos:

- **Movimento Real**: Um "estado verdadeiro"é criado com uma posição inicial (100, 50) e uma velocidade diagonal constante. Este representa o caminho perfeito do avião inimigo.
- **Medições Ruidosas**: A cada passo, o script gera uma medição da posição que é intencionalmente imprecisa (com ruído adicionado), simulando os dados do ambiente do jogo.
- **Filtro de Kalman**: O filtro é inicializado sem conhecimento prévio da velocidade real do objeto (as velocidades v_x e v_y começam em zero). A cada passo, ele recebe a medição ruidosa e aplica seu ciclo de predição e correção para estimar o estado completo do objeto (posição e velocidade).

O ponto crucial da simulação era usar a velocidade estimada (v_y) a cada passo para calcular a previsão de onde o objeto cruzaria a linha $y = 0$, conforme exibido no gráfico da Figura 9.

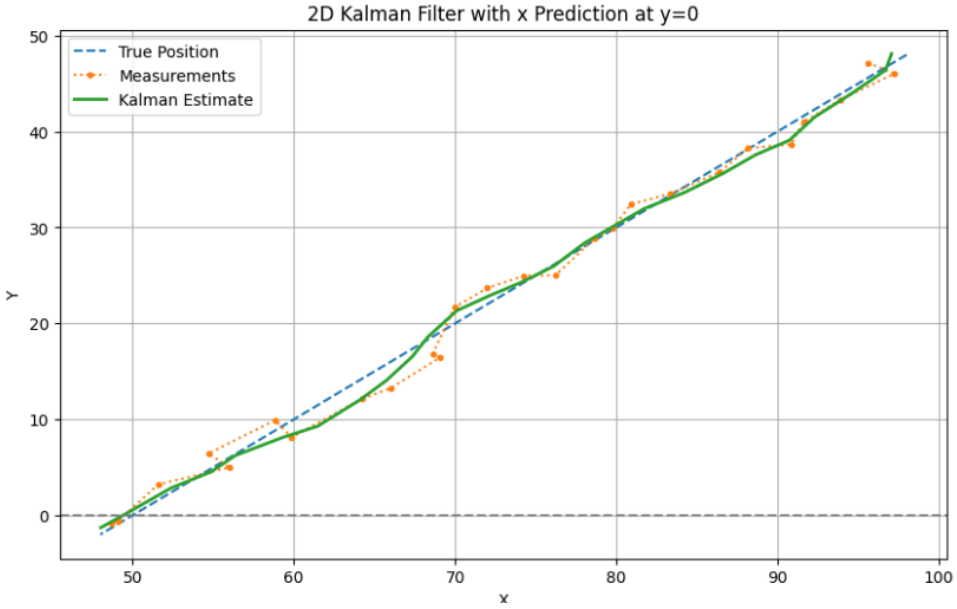


Figura 9: *Gráfico de convergência Kalman*

Como a imagem demonstra, mesmo partindo de medições ruidosas (pontos laranjas), a estimativa do Filtro de Kalman (linha verde) converge rapidamente e segue com alta fidelidade a trajetória real (linha azul tracejada). Essa validação garantiu a confiança necessária para prosseguir com a implementação, sabendo que o algoritmo era robusto o suficiente para o problema.

2.3.3 Implementação da previsão com Filtro de Kalman

A implementação foi realizada adicionando uma instância do Filtro de Kalman a cada objeto da classe Plane (avião inimigo). O filtro foi configurado para rastrear um estado composto por quatro variáveis: [posição x, posição y, velocidade v_x, velocidade v_y].

Com essa informação, o *Bot* pode calcular a coordenada x exata em que o avião atingirá a parte inferior da área de jogo, definindo então a zona de impacto. Toda vez que o jogo atualiza a posição do avião, o filtro executa seu ciclo definido no Algoritmo 2.

Algorithm 2 Fluxo de decisão para de se o avião irá colidir

1: Prediz a nova posição e velocidade com base na estimativa anterior;
2: Corrige essa predição usando a nova posição (x, y) medida na tela;
3: **if** $|\text{player.x} - \text{plane.predicted\_x\_at\_y0}| < 70$ **then**
4: $\quad plane\_will\_crash \leftarrow$ plane

Com isso, a implementação mudou radicalmente a capacidade de sobrevivência do *Bot*. Armado com a previsão do ponto de impacto, o agente agora proativamente acelera para fugir da zona de colisão, possibilitando:

- Redução de Colisões: A taxa de colisões com os aviões laterais, que era uma das principais causas de falha, foi reduzida praticamente a zero.
- Desvio com antecedência: Não espera o avião se aproximar para reagir, se move para uma zona segura assim que a trajetória do inimigo é estimada.
- Movimentos Suaves: As reações se tornaram mais estratégicas e menos bruscas.

Essencialmente, o Filtro de Kalman deu ao *Bot* a capacidade de "olhar para o futuro", transformando um problema complexo de perseguição em um cálculo de evasão direto, o que aumentou significativamente sua performance e robustez.

2.4 Q-LEARNING

O Q-Learning é uma técnica de Aprendizado por Reforço [4] que possibilita que um agente aprenda a tomar decisões de forma autônoma ao interagir com o ambiente. Trata-se de um método baseado em tentativa e erro, no qual o agente busca maximizar as recompensas obtidas por suas ações, ajustando seu comportamento com base nas experiências acumuladas ao longo dos episódios (iterações) que passaram.

O agente não possui, inicialmente, conhecimento sobre o ambiente ou sobre quais ações são mais vantajosas. Ao longo do tempo, ele aprende a identificar padrões, reagir a situações específicas e tomar decisões que aumentem sua pontuação ou o mantenham vivo por mais tempo.

A implementação do Q-Learning neste *Bot* tem como objetivo permitir que ele evolua progressivamente em suas estratégias de movimentação e ataque, sem a necessidade de regras explícitas programadas que podem entrar em conflito. O agente aprende com a prática, baseado nas consequências das ações que ele mesmo executa.

2.4.1 Funcionamento do Q-Learning

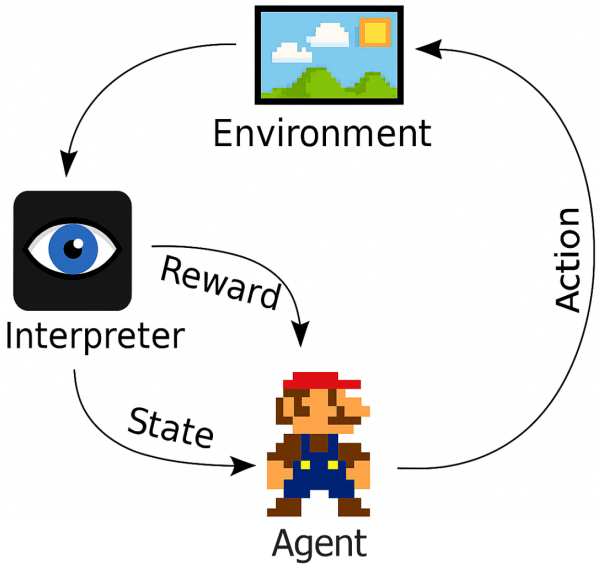


Figura 10: *Diagrama representativo do Q-Learning*

A figura ilustra o funcionamento básico do Q-Learning por meio de um ciclo de interação entre o Agente, o Ambiente, o Intérprete e os elementos de Ação, Estado e Recompensa:

- **Agente** (Agent): É o *Bot* em si, responsável por tomar decisões dentro do jogo. Escolhe uma **Ação** com base na sua política de aprendizado.
- **Ação** (Action): É a decisão tomada pelo agente, como mover-se para qualquer direção ou atirar. Influencia diretamente o futuro do **Ambiente**.
- **Ambiente** (Environment): Representa o jogo, onde ocorrem todas as interações observadas pelo **Interpretador**. O ambiente responde às ações do agente, alterando o cenário (movendo inimigos ou mudando a posição do personagem, por exemplo).
- **Interpretador** (Interpreter): É onde são definidas as regras do jogo. Responsável por avaliar a situação do Ambiente e gerar **Estado** e **Recompensa** correspondentes.

- **Estado** (State): Refere-se à representação da situação atual do ambiente, como a posição do agente na tela e a presença de obstáculos. O **Agente** utiliza essa informação para decidir sua próxima ação, baseado tabela de estados.
- **Recompensa** (Reward): Após realizar uma ação, o **Agente** recebe uma recompensa de acordo com o resultado obtido. Por exemplo, destruir um inimigo pode gerar uma recompensa positiva, enquanto colidir com um obstáculo pode resultar em uma penalidade.

Esse ciclo se repete continuamente: o agente escolhe uma ação com base no estado atual, recebe uma recompensa e atualiza sua política de comportamento. Com o tempo, ele aprende a maximizar as recompensas acumuladas, evoluindo sua capacidade de navegar e sobreviver no jogo de forma mais eficiente.

2.4.2 Implementação da tomada de decisão com Q-Learning

Um agente de Q-Learning é criado e configurado com um conjunto de ações possíveis. O agente escolhe qual ação tomar com base no estado atual do ambiente. É utilizada uma política ε-*greedy* (*Exploration/Exploitation*) para isso. A ideia é começar experimentando as diferentes possibilidades para obter as recompensas e, com o passar do tempo, começar a agir de acordo com as melhores ações para cada estado possível. O Algoritmo 3 demonstra o processo.

Algorithm 3 Pseudocódigo do fluxo de implementação do Q-Learning no programa

```
1: agent ← QLearningAgent(COMMAND_COMBOS)
2: agent.load_progress(file_path)
3: bot ← Bot()
4: while ¬bot.quit do
5:     frame ← get_frame(game)
6:     action ← agent.choose_action(state)
7:     (reward, next_state) ← bot.refresh(frame, action)
8:     agent.update(state, action, reward, next_state)
9:     state ← next_state
10: agent.save_progress(file_path)
```

A *Q-table* é uma tabela de valores interna usada no algoritmo de Q-Learning para armazenar o conhecimento que o agente adquire sobre o ambiente. Ela mapeia estados e ações para valores de recompensa esperada, ou seja, representa o quão viável é realizar cada ação em um determinado estado. Visando os conhecimentos adquiridos em execuções anteriores, o programa salva seu progresso ao final de cada execução. Ao ser reiniciado, retoma a partir desse ponto salvo, o que permite uma evolução contínua e mais eficiente do processo. Essa abordagem não apenas reduz o retrabalho, como também otimiza o desempenho ao longo do tempo.

2.4.3 Adicionando instintos ao Q-Learning

Nem sempre os estados apresentam reações imediatas às ações do agente, o que pode levá-lo a repetir diversas vezes um mesmo erro até perceber que determinada decisão não é adequada, especialmente quando situações semelhantes no passado resultaram em altas recompensas. Essa ambiguidade pode comprometer o processo de aprendizado.

Uma possível solução para mitigar esse problema é incorporar elementos do modelo de Campo Potencial à programação do agente, utilizando penalidades adicionais para desencorajar ações que já são conhecidamente arriscadas ou perigosas. Dessa forma, o agente pode ser guiado de maneira mais eficiente, evitando escolhas prejudiciais mesmo quando a recompensa imediata não for suficiente para indicar o risco.

O melhor exemplo para isso são as paredes do cenário, que não permitem toques, mesmo que mínimos. Foram definidos quatro pontos ao redor do jogador para identificar se a cor corresponde ao azul da área navegável, assim como ilustra a Figura 11. Caso algum dos pontos do mesmo lado indique uma possível colisão, a função responsável pela movimentação bloqueia deslocamentos laterais para essa direção.

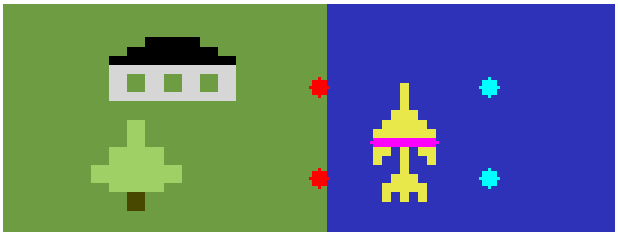


Figura 11: *Pontos indicando de colisão iminente*

3 CONCLUSÃO

Com a realização de todas as etapas do trabalho, a experiência de desenvolver um *Bot* para jogar o jogo River Raid foi extremamente enriquecedora, tanto do ponto de vista técnico quanto do aprendizado prático. Ao longo do projeto, surgiram desafios reais de percepção, tomada de decisão e adaptação em um ambiente dinâmico, que exigiram o uso de diferentes técnicas.

A aplicação de campo potencial permitiu que o agente reagisse eficientemente a obstáculos e objetivos no ambiente, como inimigos, áreas navegáveis e combustível. No entanto, situações específicas como a detecção de aviões vindos lateralmente exigiram uma abordagem mais robusta, levando à incorporação do Filtro de Kalman, que permitiu prever o movimento desses objetos e melhorar a evasão.

Além disso, a inclusão de *Q-Learning* trouxe uma camada de aprendizado adaptativo ao agente, permitindo que ele otimizasse suas ações em cenários complexos e aprendesse algumas estratégias. Ao aplicar essa camada de aprendizado, percebeu-se que os resultados não eram tão eficientes em comparação com a movimentação com campo potencial, onde ele apenas otimizou seu aprendizado para lidar com o cenário ao longo das iterações de maneira sutil.

Concluir esse projeto nos mostrou como a construção de *Bots* vai muito além da simples automação de ações: trata-se de entender profundamente o ambiente, aplicar algoritmos que auxiliam na implementação, integrando múltiplas técnicas para alcançar comportamentos mais próximos da inteligência humana. Foi uma experiência gratificante e desafiadora, onde reforçou a importância de experimentos práticos no estudo da inteligência artificial com a aplicação de *Bots* no mundo real atual.

REFERÊNCIAS

[1] Retro Game Wiki, “River Raid.” https://retro-game.fandom.com/wiki/River_Raid, 2025. Accessed: 2025-07-12.

[2] D. Dominic, “A Beginner’s Guide to Understand the Color Models RGB and HSV.” https://medium.com/@dijdomv01/a-beginners-guide-to-understand-the-color-models-rgb-and-hsv-244226e4b3e3, 2024. Published October 6, 2024; accessed July 12, 2025.

[3] W. Franklin, “Kalman Filter Explained Simply.” https://thekalmanfilter.com/kalman-filter-explained-simply/, 2020. Published December 31, 2020; accessed July 12, 2025.

[4] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.