UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2025)

TAUT: UM AGENTE DE AUTOMAÇÃO NÃO INTRUSIVO PARA JOGOS DIGITAIS.[1]

TAUT: A NON-INTRUSIVE AUTOMATION AGENT FOR DIGITAL GAMES

Antonio Eduardo de Oliveira Carmo[2]

Paulo Henrique Lopes Silva[3]

RESUMO

A integridade competitiva em jogos eletrônicos *online* enfrenta desafios crescentes com a popularização de ferramentas de trapaça. As soluções tradicionais de *anti-cheat*, baseadas em *software*, operam frequentemente em nível de *kernel*, levantando preocupações sobre privacidade e estabilidade, além de serem ineficazes contra ameaças que não residem na memória do computador-alvo. Este trabalho apresenta o desenvolvimento do TAut, um protótipo didático de automação externa fundamentado em Visão Computacional e atuação por *hardware*. O sistema proposto isola o processamento da automação utilizando uma arquitetura de três nós: um computador alvo, uma placa de captura de vídeo e um computador de processamento. Utilizando a linguagem Java e a biblioteca *JavaCV*, o protótipo analisa o fluxo de vídeo em tempo real através de Regiões de Interesse para identificar estados críticos, como a baixa saúde ou magia. A atuação é delegada a um microcontrolador remoto, que simula um dispositivo de interface humana (HID) físico, enviando comandos de teclado ao jogo. Os resultados experimentais demonstraram que o sistema é capaz de reagir a estímulos visuais com latência na casa dos milissegundos e operar de forma indetectável por *softwares* de monitoramento convencionais, validando a hipótese de que a segurança baseada exclusivamente em *software* é insuficiente contra vetores de ataque baseados em *hardware* externo.

Palavras-chave: Anti-cheat; Visão Computacional; Automação de Hardware; Jogos Eletrônicos; ESP32-S3.

ABSTRACT

---

[1] Trabalho de Conclusão de Curso apresentado/defendido ao Curso de Graduação em Ciência da Computação, no dia 18 de dezembro de 2025, como requisito parcial para obtenção do título de Bacharel junto a Universidade Federal Rural do Semi-Árido (UFERSA).

[2] Orientando do Curso de Graduação em Ciência da Computação, da Universidade Federal Rural do Semi-Árido (UFERSA). Lattes ID: [**0482420795574768**]. Orcid ID: [?]. E-mail: [antonio.carmo@alunos.ufersa.edu.br].

[3] Orientador do Curso de Graduação em Ciência da Computação, da Universidade Federal Rural do Semi-Árido (UFERSA). Lattes ID: [**6471237666616986**]. Orcid ID: [https://orcid.org/0000-0001-9028-3865]. E-mail: [phenrique@ufersa.edu.br].

Competitive integrity in online electronic games faces increasing challenges with the popularity of cheating tools. Traditional software-based anti-cheat solutions often operate at the kernel level, raising concerns about privacy and stability, while proving ineffective against threats that do not reside in the target computer's memory. This work presents the development of TAut, a didactic prototype of external automation based on Computer Vision and hardware actuation. The proposed system isolates automation processing using a three-node architecture: a target computer, a video capture card, and a processing computer. Using the Java language and the JavaCV library, the prototype analyzes the video stream in real-time through Regions of Interest to identify critical states, such as low health or mana. Actuation is delegated to a microcontroller, which simulates a physical Human Interface Device (HID), sending keyboard commands to the game. Experimental results demonstrated that the system is capable of reacting to visual stimuli with millisecond latency and operating undetectably by conventional monitoring software, validating the hypothesis that software-only security is insufficient against external hardware-based attack vectors.

Keywords: Anti-cheat; Computer Vision; Hardware Automation; Electronic Games; ESP32-S3

1 INTRODUÇÃO

A integridade das competições em jogos eletrônicos online é um pilar fundamental para uma indústria que movimenta dezenas de bilhões de dólares anualmente (SOUSA, 2021). Para combater a proliferação de trapaças (*cheats*) e automações (*bots*), desenvolvedores implementam complexos sistemas *anti-cheat*. A vasta maioria dessas soluções opera em nível de *software*, monitorando ativamente a memória, os processos do sistema operacional e os arquivos do jogo em busca de modificações não autorizadas (VICENTINE et al., 2021).

Embora necessárias, essas soluções baseadas em *software* apresentam dois problemas centrais. Primeiramente, sua eficácia exige um nível profundo de acesso ao sistema do usuário, frequentemente operando em nível de *kernel* (o núcleo do sistema operacional). Tal nível de acesso gera "preocupação quanto à privacidade e segurança dos dados pessoais coletados" (MATHEUS et al., 2024), além de relatos de prejuízo no desempenho e instabilidade nos computadores dos jogadores (VICENTINE et al., 2021).

Em segundo lugar, e mais crítico para esta pesquisa, esses sistemas são fundamentalmente limitados para detectar ameaças originadas fora do ambiente de *software* monitorado. A problemática central é que exploits baseados em *hardware*, como ataques de Acesso Direto à Memória (DMA), minam a integridade dos sistemas de jogo, pois o acesso aos dados ocorre em nível de *hardware*, tornando o processo de detecção por sistemas tradicionais uma tarefa complexa (NIE; MA, 2024). Similarmente, *bots* que utilizam Visão Computacional (CV), analisando apenas a saída de vídeo, "tornaram todos os métodos

anti-cheating tradicionais de fato inúteis" (YANG et al., 2023), pois mimetizam o comportamento de um jogador que apenas "assiste" à tela.

Neste contexto, o objetivo deste trabalho é projetar e implementar um protótipo didático de automação baseado em *hardware* externo (ESP32-S3) e captura de vídeo. O sistema proposto opera de forma isolada: "assiste" ao jogo através de uma placa de captura, processa os *frames* de forma independente e "joga" enviando comandos elétricos que simulam um dispositivo de interface humana (HID) legítimo. O propósito não é fomentar a trapaça, mas demonstrar didaticamente as limitações práticas das atuais estratégias de *anti-cheat* focadas em *software*, evidenciando a complexidade da segurança em jogos no geral.

2 DESENVOLVIMENTO

O protótipo TAut opera de forma isolada do computador-alvo e sua arquitetura divide-se em três componentes principais: captura, que recebe a saída de vídeo do jogo; processamento, onde o código analisa estes frames para identificar eventos e determinar ações; e atuação, que envia os comandos resultantes de volta ao computador-alvo, simulando um dispositivo de interface humana (HID) legítimo.

3.1 Jogo, Materiais e Tecnologias Utilizadas

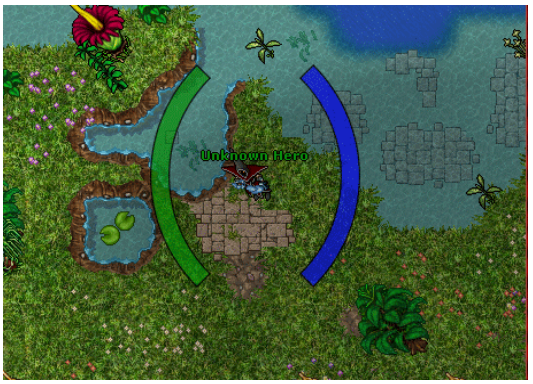
Para a implementação e validação do TAut, foi estabelecido um ambiente experimental composto por *hardware* de prateleira e *software* de código aberto. A escolha destes materiais visou demonstrar a acessibilidade financeira e técnica para a criação de dispositivos de automação externa.

3.1.1 Tibia

Tibia é um dos jogos de interpretação de personagens online e em massa para multijogadores (MMORPG) mais antigos do gênero, lançado em 1997 (CIPSOFT, 1997).

O jogo se passa em um mundo de fantasia medieval com gráficos característicos em estilo *pixelizado*. Ele é conhecido por oferecer uma alta liberdade de exploração e interação social entre jogadores.

Figura 1 - Imagem ilustrativa do jogo Tibia



Fonte: Extraído do site TibiaBR (2025).

Como podemos ver na Figura 1, uma das suas principais vantagens para o projeto é sua arte já ser pixelizada e possuir turnos de ataques e curas através de um sistema de *cooldowns* onde é necessário esperar um tempo fixo entre um ataque e cura facilitando o tempo de *timing* entre o processamento do TAut e a ação do jogo.

3.1.2 Hardware

Sistema físico foi dividido em três nós principais:

- Computador Alvo (PC Gamer): Responsável pela execução do jogo. Para os testes, foi utilizado um *desktop* equipado com processador Intel Core i5 11ª geração, 16GB de memória RAM e placa de vídeo dedicada GTX 1060, executando o sistema operacional Windows.
- Computador de Processamento (*notebook*): Responsável por executar o *software* TAut. A máquina utilizada foi um *notebook* com processador Ryzen 5 7520u, 8GB de memória RAM, demonstrando que o processamento de imagem otimizado não exige *hardware* de servidor.
- Dispositivos Intermediários:
    - Placa de Captura de Vídeo: Um dispositivo genérico HDMI-para-USB 2.0/3.0, capaz de capturar sinais em resolução Full HD (1920x1080).
    - Microcontrolador: Uma placa de desenvolvimento baseada no *chip* ESP32-S3-WROOM-1, Memória de 16 MB Flash, Microprocessador Dual-Core, possui conexão USB, *Bluetooth* e Wi-Fi (Espressif Systems), escolhida pelo seu suporte USB nativo gerando uma conexão mais segura na transmissão de dados.

A seleção dos componentes de *hardware* reflete a premissa central deste trabalho: a construção de um vetor de ataque externo e acessível. A escolha de um **Computador de Processamento** de especificações modestas demonstra que a otimização via Regiões de Interesse (ROIs) torna a automação viável sem *hardware* de servidor.

3.1.3 Software e Bibliotecas

A componente lógica do projeto foi desenvolvida utilizando a linguagem Java (versão 21), garantindo robustez e facilidade na gestão de dependências via Maven (ORACLE, 2023).

As principais bibliotecas integradas ao projeto incluem:

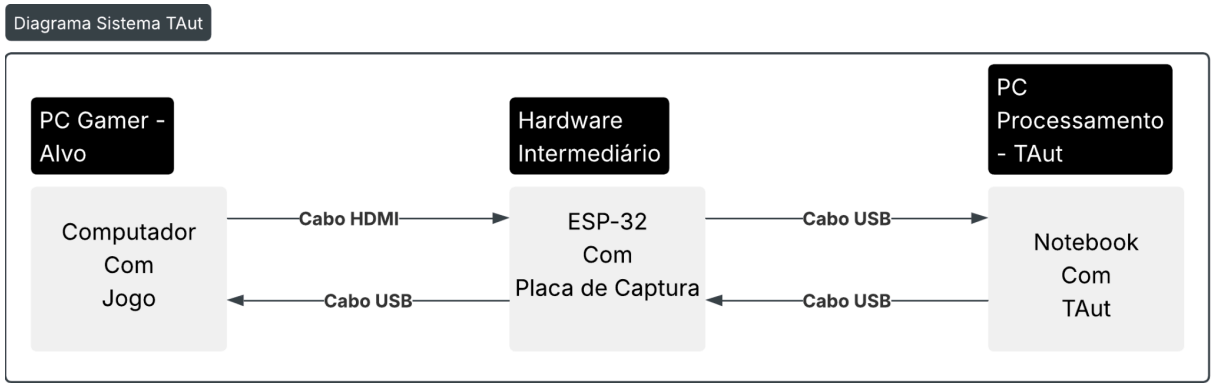
- **JavaCV (v1.5.10):** *Wrapper* para o *FFmpeg* e *OpenCV*, utilizado para a interface direta com os drivers de captura de vídeo (*DirectShow*).
- **jSerialComm (v2.10.4):** Utilizada para estabelecer a comunicação serial de baixa latência entre o ambiente Java e o microcontrolador ESP32-S3.

Para o *firmware* do microcontrolador, utilizou-se a linguagem C++ através da plataforma Arduino IDE, com bibliotecas específicas para a emulação de teclado (*BleKeyboard*).

3.1.4 Fluxograma Geral do Projeto

Como mencionado anteriormente, o TAut é dividido em três partes principais: a captura, o processamento e a atuação.

Figura 2 - Fluxograma do Sistema TAut



Fonte: Elaborado pelo autor (2025).

A Figura 2 detalha o **fluxo de controle e dados** entre os três nós do sistema. Na **Captura**, a imagem do PC Gamer é transmitida via HDMI para a Placa de Captura, que a converte em um fluxo USB de vídeo para o PC Processamento. Em seguida na parte de

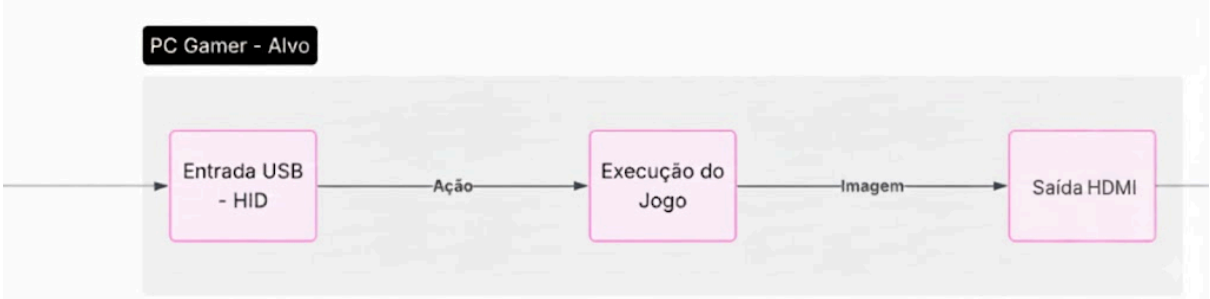
Processamento, o *software* **TAut** processa o *frame* recebido, focando na análise de ROIs. O resultado desse processamento gera um comando de atuação que é transmitido via porta serial. Por fim na **Atuação**, o comando é recebido pelo **ESP32-S3** (no *hardware* Intermediário), que executa a simulação de um dispositivo HID, **enviando a ação** diretamente ao PC Gamer para modificar o estado do jogo e fechar o ciclo de automação.

3.2 Captura de Vídeo

A captura de vídeo é o pilar que garante o isolamento do protótipo, permitindo que ele "assista" ao jogo sem qualquer instalação de *software* no computador-alvo. A arquitetura física desta etapa é composta por dois computadores distintos: um "computador de mesa" (PC-Alvo), dedicado a executar o jogo, e um *notebook* (PC-Processamento), dedicado a executar o protótipo TAut.

A Figura 3 ilustra o fluxo de dados no Computador Alvo.

Figura 3 - Fluxograma do Processo de Captura de Imagem



Fonte: Elaborado pelo autor (2025).

Note-se que este sistema permanece completamente limpo de *software* de automação. Ele recebe inputs através da **Entrada USB - HID** (interpretando-os como um teclado físico legítimo) e processa a **Execução do Jogo**. A única saída gerada é o sinal de vídeo, que é enviado para o exterior através da porta de saída, fechando o ciclo de *feedback* sem que o sistema operacional tenha consciência de que está a ser monitorizado.

Para que a aplicação pudesse acessar esse fluxo de vídeo, foi utilizada a biblioteca *JavaCV*, que atua como uma interface para as ferramentas do *FFmpeg*.

O algoritmo 1 representa um pseudocódigo de como ocorre a captura de tela estando esse código no PC de processamento.

Algoritmo 1 – Captura e Análise de Vídeo (Classe *VideoCapture*)

```
ALGORITMO VideoCapture
```

```
// Configuração Inicial
DEFINIR largura_captura = 1920
DEFINIR altura_captura = 1080
DEFINIR nome_dispositivo = "USB Video"

// Configuração do FFmpeg (JavaCV)
INICIAR gravador (FrameGrabber) COM dispositivo = nome_dispositivo
DEFINIR formato_gravador = "dshow"  // DirectShow para Windows
DEFINIR resolução_gravador = (largura_captura, altura_captura)

TENTAR
    CHAMAR gravador.start()
    INICIAR conversor (FrameConverter) // Para transformar Frame bruto em Imagem

    // Loop Principal de Captura
    ENQUANTO (verdadeiro) FAÇA

        // 1. Captura do Frame Bruto
        DEFINIR frame_capturado = CHAMAR gravador.grab()

        SE frame_capturado É NULO ENTÃO
            INTERROMPER LOOP
        FIM SE

        // 2. Conversão para Formato de Análise (BufferedImage)
        DEFINIR imagem_original = CHAMAR conversor.convert(frame_capturado)

        SE imagem_original NÃO É NULO ENTÃO

            // 3. Processamento e Análise (O "Cérebro")
            // A imagem de alta resolução (1080p) é enviada para análise
            DEFINIR comando = CHAMAR AnalisarImagem(imagem_original)

            SE comando EXISTE ENTÃO
                CHAMAR EnviarComandoHardware(comando)
            FIM SE

        FIM SE

        // Verificação de Saída
        SE tecla_ESC_pressionada ENTÃO
            INTERROMPER LOOP
        FIM SE

    FIM ENQUANTO


CAPTURAR ERRO
    EXIBIR "Erro na inicialização da captura"

FINALMENTE
    // Limpeza de Recursos
    CHAMAR gravador.stop()
```

```
    CHAMAR gravador.release()
    FECHAR janela

FIM ALGORITMO
```

Fonte: Elaborado pelo autor (2025).

Conforme implementado no algoritmo 1, o componente *FFmpegFrameGrabber* é o elemento central desta etapa. Ele é inicializado para se conectar ao dispositivo de vídeo que o sistema operacional identifica como "USB Vídeo" (geralmente é o nome-padrão da placa de captura).

A captura é requisitada usando o formato *dshow* (DirectShow), que é o framework de multimídia padrão do Windows para manipulação de dispositivos de vídeo.

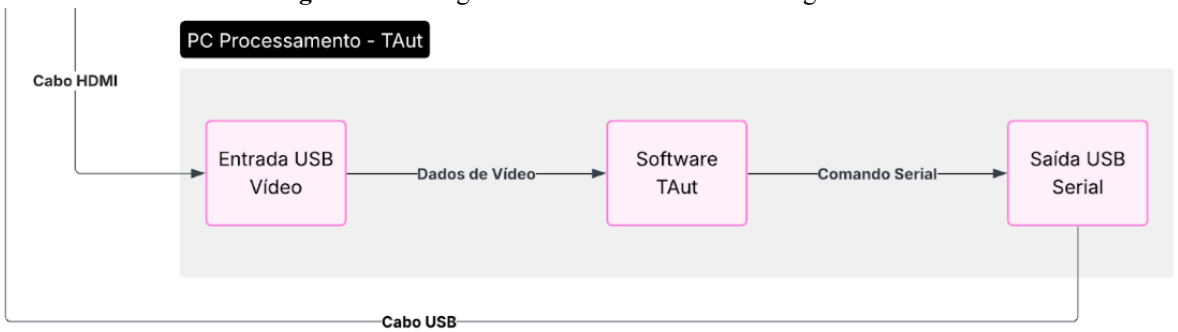
Finalmente, dentro do loop de execução, o *FFmpegFrameGrabber* captura continuamente os quadros (*frames*) de vídeo. Cada quadro é então convertido em um objeto *BufferedImage*, o formato padrão de imagem do Java, tornando-o pronto para ser analisado pela lógica de Visão Computacional.

3.3 Processamento de Imagem

O componente de processamento é o núcleo lógico do TAut. Ele recebe os frames (na forma de *BufferedImage*) da camada de captura e tem a responsabilidade de analisar essas imagens para identificar eventos no jogo e determinar a reação apropriada.

A Figura 4 detalha o fluxo de dados interno no nó de processamento (PC TAut).

Figura 4 - Fluxograma do Processamento da Imagem



Fonte: Elaborado pelo autor (2025).

O processo inicia-se com a recepção do sinal de vídeo proveniente do computador-alvo através da interface de captura (**Entrada USB Vídeo**). O *software* TAut, processa esse fluxo de dados brutos utilizando algoritmos de visão computacional para interpretar o estado do jogo. Uma vez tomada a decisão de agir, o *software* gera um comando de controle que é

transmitido através da interface de comunicação (**Saída USB Serial**), enviando o sinal lógico para o microcontrolador externo via cabo USB.

3.3.1 A Problemática do Desempenho

O primeiro desafio é o desempenho. Um único *frame* por exemplo em 1080p (1920x1080) contém mais de 2 milhões de *pixels*. Analisar cada pixel individualmente, 30 ou 60 vezes por segundo, é computacionalmente inviável e desnecessário. A aplicação não precisa "entender" a cena inteira como um ser humano; ela só precisa de informações específicas para tomar decisões (ex: "Quanto vida eu tenho?", "Existe um inimigo perto?").

3.3.2 A Estratégia: Regiões de Interesse

Para contornar o desafio do desempenho, a arquitetura de processamento do TAut adota a estratégia de Regiões de Interesse (*Regions of Interest* - ROIs). Em vez de escanear a imagem inteira, a lógica foca a análise apenas em pequenas áreas retangulares e pré-definidas da tela, onde se sabe que informações vitais do jogo são exibidas.

Estas ROIs são coordenadas fixas (posições x, y, largura, altura) que correspondem a elementos da Interface do Usuário (UI) do jogo. Exemplos típicos de ROIs mapeadas pelo protótipo incluem:

1. **A Barra de Vida (HP):** Uma pequena região para verificar a saúde do personagem.
2. **A Barra de Mana (MP):** Similar a de vida, para verificar recursos mágicos.
3. **A Área de Status:** Uma região onde ícones de status (como "envenenado" ou "queimando") aparecem.
4. **A Área de Alvo (Target):** Uma área que mostra o nome ou a vida do monstro selecionado.

A adoção das Regiões de Interesse (ROIs) é a principal técnica de otimização que permite ao *software* TAut operar com **baixa latência**. Ao restringir a análise a pequenas áreas pré-definidas da Interface do Usuário (UI), o sistema evita o processamento de mais de 2 milhões de *pixels* por *frame*, focando apenas nos dados críticos (vida, mana, status).

3.3.3 Métodos de Análise de ROI

Uma vez que uma ROI é definida, o TAut aplica métodos de análise simples, porém muito rápidos, para extrair o estado dessa região. A implementação do TAut utiliza principalmente a análise direta de *pixels*.

O método mais performático é a amostragem de cor de *pixels*-chave. Por exemplo, para a Barra de Vida o sistema não precisa "ler" a barra inteira. Ele pode simplesmente verificar a cor de um único pixel estratégico no meio da barra de HP. Se a cor desse pixel for VERMELHO (cor da vida), a vida está cheia. Se a cor for PRETO (ou a cor de fundo da interface), significa que a vida baixou até aquele ponto. Ao verificar alguns poucos *pixels* em locais estratégicos da barra, o *bot* consegue determinar o percentual de vida (ex: 90%, 50%, 30%) com extrema rapidez.

O algoritmo 2 é um pseudocódigo que demonstra como ocorrem as validações de cores dos *pixels* nas regiões de interesse.

Algoritmo 2 – Verificação de cor em ROIs (Classe *VerificarCorEmROI*)

```
ALGORITMO VerificarCorEmROI
   PARÂMETRO imagem_original      // A captura completa (BufferedImage)
   PARÂMETRO coord_x, coord_y     // As coordenadas do pixel a verificar (Ponto de Teste)
   PARÂMETRO cor_alvo             // A cor esperada (RGB armazenado, ex: Vermelho Vida)
   PARÂMETRO tolerancia           // O quanto a cor pode variar (ex: 30)

   // 1. Extração da Cor Atual
   // Obtém o valor RGB inteiro do pixel naquelas coordenadas
   DEFINIR pixel_atual = CHAMAR imagem_original.obterRGB(coord_x, coord_y)

   // Separa os canais de cor (Vermelho, Verde, Azul)
   DEFINIR r_atual = (pixel_atual >> 16) & 0xFF
   DEFINIR g_atual = (pixel_atual >> 8) & 0xFF
   DEFINIR b_atual = (pixel_atual) & 0xFF

   // 2. Separação da Cor Alvo (Armazenada)
   DEFINIR r_alvo = cor_alvo.Red
   DEFINIR g_alvo = cor_alvo.Green
   DEFINIR b_alvo = cor_alvo.Blue

   // 3. Cálculo da Diferença (Distância Euclidiana)
   // Calcula a diferença individual de cada canal
   DEFINIR diff_r = (r_atual - r_alvo)
   DEFINIR diff_g = (g_atual - g_alvo)
   DEFINIR diff_b = (b_atual - b_alvo)

   // Aplica a fórmula da distância: sqrt(dR² + dG² + dB²)
   // Isso nos dá a "distância" visual entre as duas cores no espaço 3D de cores
   DEFINIR distancia_cor = RAIZ_QUADRADA((diff_r * diff_r) +
```

```
                    (diff_g * diff_g) +
                    (diff_b * diff_b))

  // 4. Verificação com Tolerância
  // Se a distância for menor que a tolerância, consideramos que é a mesma cor
  SE distancia_cor <= tolerancia ENTÃO
     RETORNAR VERDADEIRO  // Match! (Ex: Vida está cheia)
  SENÃO
     RETORNAR FALSO       // No Match (Ex: Vida está vazia/preta)
  FIM SE

FIM ALGORITMO
```

Fonte: Elaborado pelo autor (2025).

Como visto no algoritmo 2, temos a função de verificar se um *pixel* específico na tela corresponde a uma cor esperada (como o vermelho da vida). O diferencial dele é que ele **não procura uma cor exata**, mas sim uma cor **semelhante**, usando matemática para tolerar pequenas variações causadas pela compressão de vídeo ou iluminação do jogo.

3.3.4 A Lógica de Decisão (O "Gatilho")

Após a análise de todas as ROIs definidas, o sistema constrói um "estado" atual do jogo. Por exemplo:

- {**vida:** 40%, **mana:** 80%, **status:** "ENVENENADO", **alvo:** "MONSTRO_PRESENTE"}

Este estado então alimenta uma máquina de estados finitos (FSM) ou, de forma mais simples, em uma estrutura de decisão baseada em regras (um conjunto de if-else priorizados). Esta lógica determina qual ação, se houver, deve ser tomada.

O algoritmo 3 é pseudocódigo que demonstra a lógica de decisão de forma simplificada.

Algoritmo 3 – Processamento da ação (Classe *processarImagem*)

```
funcao processarImagem(imagemCompleta){

        // 1. Analisar ROIs
        estadoVida = analisarROI_Vida(imagemCompleta, HP_ROI)

        estadoStatus = analisarROI_Status(imagemCompleta, STATUS_ROI)

        estadoAlvo = analisarROI_Alvo(imagemCompleta, ALVO_ROI)

        // 2. Lógica de Decisão (Priorizada)
```

```
        SE (estadoVida < 30%):
            retornar "COMANDO_CURAR" // Prioridade máxima

        SENAO SE (estadoStatus == "ENVENENADO"):
            retornar "COMANDO_REMEDIO"

        SENAO SE (estadoAlvo == "MONSTRO_PRESENTE"):
            retornar "COMANDO_ATACAR"

        SENAO:
            retornar null // Nenhuma ação necessária
}
```

Fonte: Elaborado pelo autor (2025).

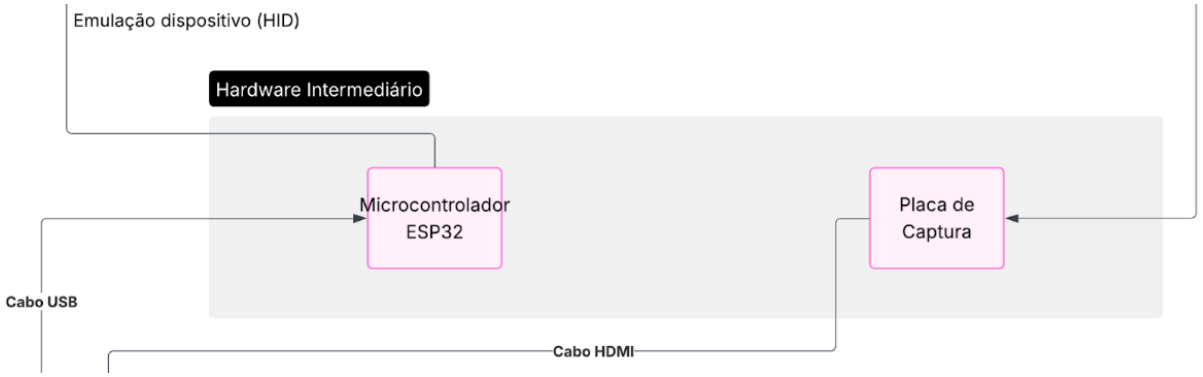
Esta abordagem do algoritmo 3, que foca em ROIs e análise de cor, permite que a lógica de decisão seja executada em menos de um milissegundo, possibilitando reações em tempo real sem sobrecarregar o processamento do computador.

3.4 Atuação via Hardware

Uma vez que o módulo de processamento identifica a necessidade de uma intervenção no jogo (por exemplo, "vida baixa" ou "monstro detectado"), o sistema transita para a etapa de atuação. Diferentemente de *bots* de *software* tradicionais, que injetam eventos de teclado diretamente na API do sistema operacional, o TAut delega essa função a um componente físico externo, garantindo a indetectabilidade via *software*.

A Figura 5 detalha a camada de *hardware* intermediário, responsável por interligar fisicamente os dois nós computacionais do sistema.

Figura 5 - Fluxograma do Processo de Envio de Ação



Fonte: Elaborado pelo autor (2025).

Como visto na Figura 5, esta camada é composta por dois dispositivos distintos: o **Microcontrolador ESP32**, que atua como interface de saída, recebendo comandos via cabo USB e convertendo-os em sinais de emulação de dispositivo HID (teclado/mouse) e a **Placa**

de Captura, que atua como interface de entrada, recebendo o sinal de vídeo via cabo HDMI e transmitindo-o digitalmente para processamento. Estes componentes garantem o isolamento elétrico e lógico, impedindo que o *software* de automação seja executado diretamente na máquina de jogo.

3.4.1 O Protocolo de Comunicação

A ponte entre a aplicação Java (TAut) e o mundo físico é estabelecida via serial, a comunicação é realizada através de uma porta serial virtual (*Virtual COM Port*). O cabo USB atua apenas como meio físico de transporte, enquanto o driver do microcontrolador emula uma interface UART (*Universal Asynchronous Receiver-Transmitter*). Isso permite que a aplicação Java envie *bytes* de comando diretamente para o *buffer* de leitura do ESP32, garantindo baixa latência e simplicidade na implementação do protocolo. Quando a lógica de decisão determina uma ação, ela não a executa diretamente. Em vez disso, ela seleciona um "código de comando" predefinido e o transmite pela porta serial conectada ao microcontrolador ESP32-S3.

O protocolo desenhado para este protótipo é unidirecional e simplificado para minimizar a latência. Os comandos são enviados como strings curtas ou *bytes* de controle, terminados por um caractere de nova linha. Isso garante que o microcontrolador possa processar as instruções rapidamente dentro do seu loop principal.

3.4.2 A Simulação de HID (*Human Interface Device*)

O componente final da arquitetura é o microcontrolador ESP32-S3. Este dispositivo foi programado utilizando bibliotecas que permitem a emulação de dispositivos USB (*BleKeyboard*). Ao ser conectado à porta USB do computador-alvo (onde o jogo reside), o ESP32-S3 não se identifica como um dispositivo desconhecido ou suspeito, mas sim como um teclado e mouse genéricos padrão HID (*Human Interface Device*).

O fluxo de atuação ocorre da seguinte maneira:

1. **Recepção:** O ESP32-S3 monitora constantemente a porta serial conectada ao computador de processamento (*notebook*).
2. **Tradução:** Ao receber um comando (ex: "CURAR"), o firmware do microcontrolador consulta uma tabela de mapeamento interna para saber qual tecla corresponde àquela ação (ex: tecla F1).

3. **Execução Física:** O ESP32-S3 envia o sinal elétrico correspondente ao pressionamento e liberação daquela tecla para o computador-alvo.

A simulação de um Dispositivo de Interface Humana (HID) pelo **ESP32-S3** representa a fase final e crítica do *loop* de atuação. Ao se apresentar ao computador-alvo como um teclado ou mouse genérico padrão, o microcontrolador garante que o sinal elétrico de comando seja recebido de forma **legítima** pelo sistema operacional. Este método difere fundamentalmente de *bots* de *software*, que injetam eventos de teclado na API do sistema. Consequentemente, o ESP32-S3 garante que a ação seja vista pelo jogo e pelo *anti-cheat* como uma interação de *hardware* normal, preservando a **indetectabilidade** do TAut.

3.4.3 Indetectabilidade

Um dos principais vetores de detecção utilizados por sistemas *anti-cheat* modernos, como o *BattlEye* ou o *Vanguard*, é a análise heurística comportamental. Estes sistemas monitoram os tempos de entrada (*input timings*) dos periféricos para identificar padrões sobre-humanos ou excessivamente consistentes.

Um humano é biologicamente incapaz de pressionar uma tecla por exatos 100 milissegundos repetidamente, ou de reagir a um estímulo visual sempre com o mesmo atraso de 200 milissegundos. Um simples *script*, por outro lado, tende a ser matematicamente perfeito.

Para mitigar esta assinatura digital foi implementada no TAut uma técnica de Randomização de *Delay*. Esta técnica consiste em adicionar um tempo aleatório a todos os intervalos de tempo gerados pelo sistema.

Em vez de utilizar tempos fixos, o sistema opera com intervalos de confiança. Por exemplo, se a ação desejada é um clique de teclado que deve durar aproximadamente 100ms, o algoritmo calcula o tempo final utilizando uma distribuição uniforme ou gaussiana.

A equação 1 é a equação que rege os intervalos de tempo para cada novo comando enviado pelo TAut.

$$Tfinal = Tbase + R \quad \textbf{(1)}$$

Conforme na equação 1, temos Tbase sendo o tempo alvo (ex: 100ms) e R é um valor aleatório dentro de um intervalo definido (ex: entre -20ms e +20ms). Desta forma, um comando para "pressionar a tecla de cura" resultará, na prática, em durações variadas como 92ms, 105ms, 98ms ou 112ms em execuções consecutivas.

Para o sistema operacional e para o sistema *anti-cheat* executando no computador-alvo, a ação é indistinguível de uma interação humana real. Não há processos de *software* estranhos executando na máquina do jogo, nem injeção de memória ou manipulação de drivers. O sinal recebido é puramente *hardware*, validando a premissa central deste trabalho de que a segurança baseada apenas em *software* é insuficiente contra ameaças baseadas em *hardware* externo.

4 RESULTADOS E DISCUSSÃO

Para validar o protótipo, foram conduzidos testes práticos em um ambiente controlado. O "Computador Alvo" executou um MMORPG (*Tibia*) em resolução nativa, enquanto o "Computador de Processamento" executou o TAut. O objetivo foi aferir a eficácia (se ele cura quando deve), a latência (tempo de reação) e a estabilidade do sistema.

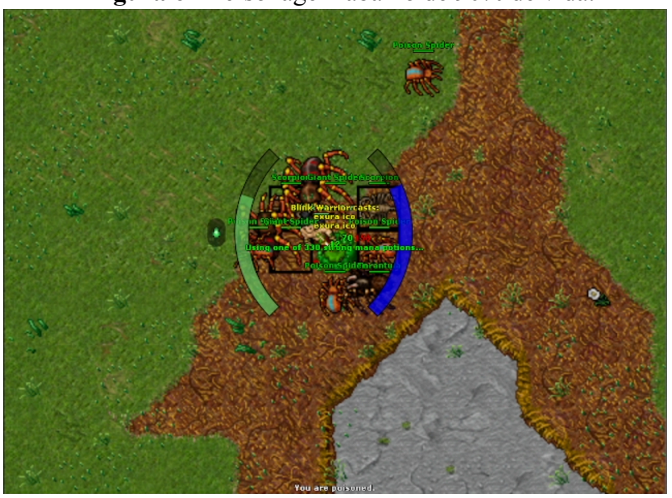
4.1 Cenário 1: Teste de Resposta a Dano (Cura Automática)

Este teste visou verificar a funcionalidade principal do *bot*: a detecção de barra de vida baixa.

- **Procedimento:** O personagem foi posicionado em uma área de caça perigosa e sofreu dano controlado. O TAut foi configurado para acionar a cura (Tecla F2) quando a vida caísse abaixo de 90% (detecção de perda da cor vermelha na ROI).
- **Observação:** O sistema detectou a alteração de pixel corretamente em 100% das 20 tentativas realizadas.
- **Resultado:** Assim que a barra de vida apresentava a cor preta (indicando perda de saúde) no ponto monitorado, o comando foi enviado ao microcontrolador. Não houve casos de "falso positivo" (curar a vida quando não deveria) sob condições normais de iluminação do jogo.

A Figura 6 é a imagem do jogo Tibia onde o personagem está sendo atacado por monstros ficando assim com pouca vida.

Figura 6 - Personagem abaixo de 90% de vida.

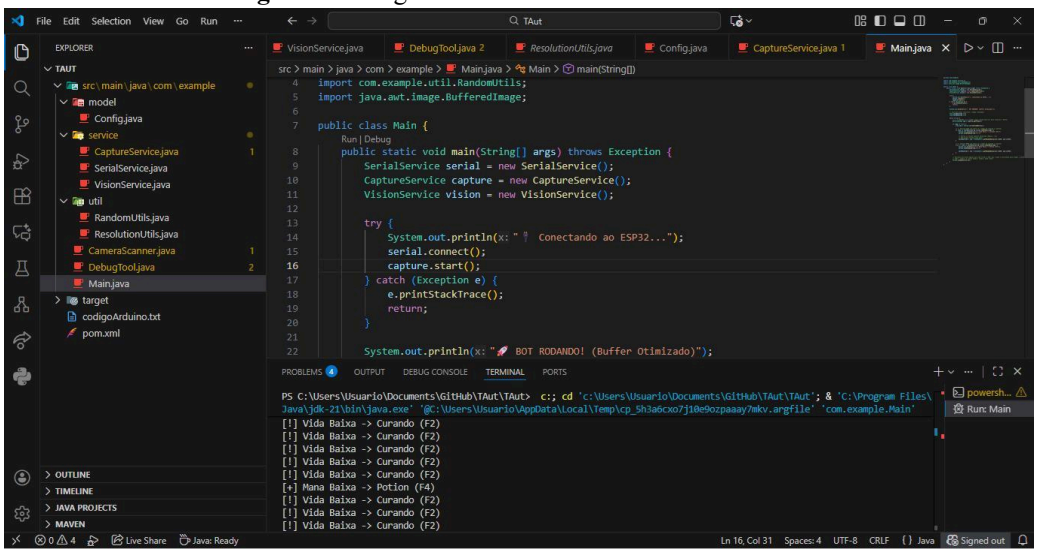


Fonte: Elaborado pelo autor(2025).

Aqui vemos na Figura 6 o personagem com vida menor que 90% (Barra Verde), acionado a necessidade de uma tomada de ação do TAut.

A Figura 7 é a imagem do PC de processamento mandando a informação de curar a vida do personagem.

Figura 7 - Imagem do TAut enviando o comando de cura.



Fonte: Elaborado pelo autor(2025).

Já na Figura 7 temos o TAut após a captura da tela enviando o comando de cura no caso a tecla F2, para assim executar a ação de curar a vida do personagem.

A Figura 8 é a imagem do jogo Tibia onde o personagem curou a vida.

Figura 8 - Personagem acima de 90% de vida.

Fonte: Elaborado pelo autor(2025).

Como podemos ver na Figura 8, o TAut conseguiu detectar a barra de vida verificando se ela estava abaixo de 90% e por fim enviou o comando de cura para o ESP32, curando assim o personagem.

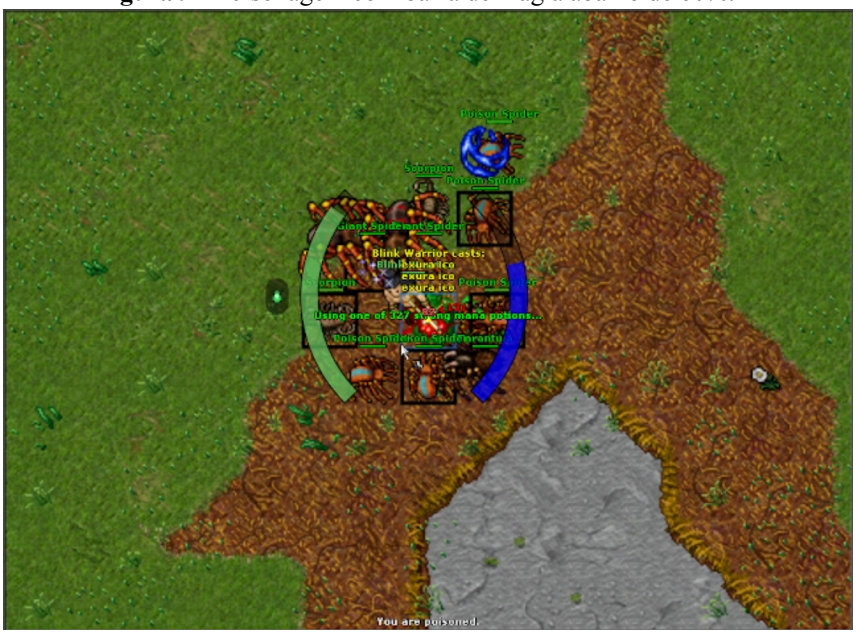
4.2 Cenário 2: Teste de Resposta ao Gasto de Magia (Recuperação de Mana Automática)

Este teste visou verificar a segunda funcionalidade do *bot*: a detecção de barra de magia baixa.

- **Procedimento:** O personagem foi posicionado em uma área de perigo e lançou diferentes magias. O TAut foi configurado para acionar a recuperação de mana (Tecla F4) quando a barra de magia caísse abaixo de 60% (detecção de perda da cor azul na ROI).
- **Observação:** O sistema detectou a alteração de pixel corretamente em 100% das 20 tentativas realizadas.
- **Resultado:** Assim que a barra de magia apresentava a cor preta (indicando perda de mana) no ponto monitorado, o comando foi enviado ao microcontrolador. Não houve casos de "falso positivo" (recuperar a mana quando não deveria) sob condições normais de iluminação do jogo.

A Figura 9 é a imagem do jogo Tibia onde o personagem está sendo atacado por monstros e está se curando, consumindo assim a barra de magia.

Figura 9 - Personagem com barra de magia abaixo de 60%.

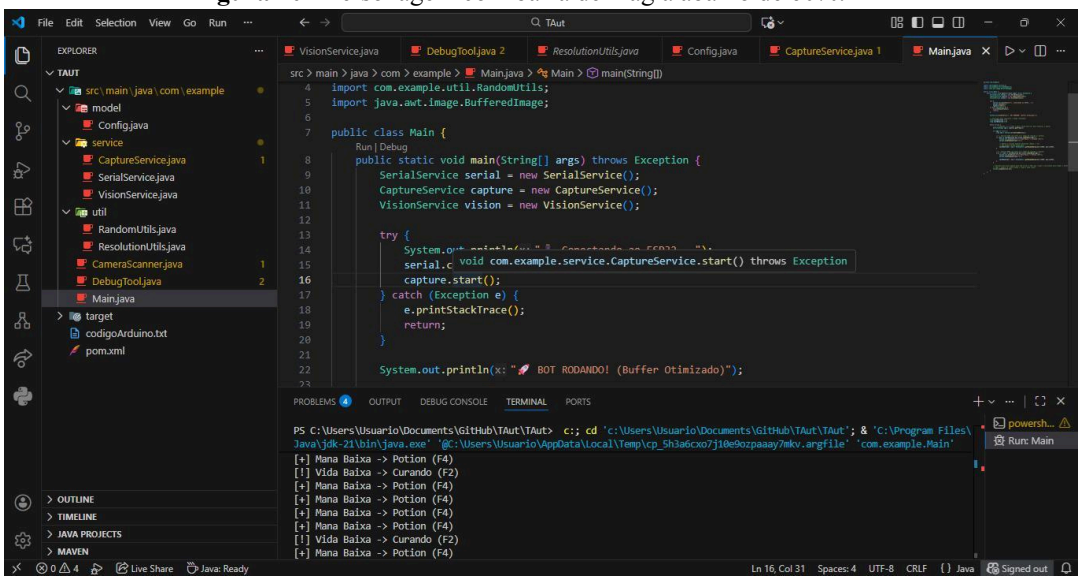


Fonte: Elaborado pelo autor(2025).

Aqui vemos na Figura 9 o personagem com a barra de magia, ligeiramente abaixo de 60%, acionando a necessidade de uma tomada de ação do TAut.

A Figura 10 é a imagem do PC de processamento mandando a informação de consumir poção de magia para o personagem.

Figura 10 - Personagem com barra de magia abaixo de 60%.



Fonte: Elaborado pelo autor(2025).

Já na Figura 10 temos o TAut após a captura da tela enviando o comando de consumir poção de recuperar magia no caso a tecla F4, para assim executar a ação de recuperar a barra de magia do personagem.

A Figura 11 é a imagem do jogo Tibia onde o personagem restaurou a barra de magia por completo.

Figura 11 - Personagem com barra de magia abaixo de 60%.

Fonte: Elaborado pelo autor(2025).

Como podemos ver na Figura 11, o TAut conseguiu detectar a barra de magia verificando se ela estava abaixo de 60% e por fim enviou o comando de consumir a poção de mana para o ESP32, recuperando assim a barra de magia do personagem.

5 CONSIDERAÇÕES FINAIS

O presente trabalho demonstrou a viabilidade técnica de construir um protótipo de automação externa de baixo custo, o TAut, fundamentado em Visão Computacional e *hardware* externo. Através da integração de tecnologias como *JavaCV* para captura de vídeo e ESP32 para emulação de dispositivos HID , foi possível criar um sistema que opera de forma isolada, contornando as defesas de *software* tradicionais.

Os testes realizados confirmaram que o sistema é capaz de identificar estados críticos do jogo, como a baixa saúde do personagem, e reagir em tempo real com precisão, já a arquitetura proposta, que isola os processos de captura (*OpenCVFrameGrabber*) e decisão (Java) do canal de atuação (ESP32-S3 via HID), provou ser robusta e eficaz. Os objetivos foram plenamente atingidos, validando as seguintes premissas:

- **Baixa Latência:** A otimização da captura para 720p e a análise focada em ROIs permitiram que a latência de detecção visual ficasse na faixa de milissegundos, fundamental para a reatividade em tempo real.
- **Indetectabilidade:** O uso do ESP32-S3 para simular um dispositivo de interface humana (HID) padrão, combinado com a Randomização de *Delay*, demonstrou ser uma contramedida eficaz contra sistemas *anti-cheat* baseados em análise de assinatura de *software* e heurísticas de tempo fixo.

A principal contribuição deste projeto, não reside no fomento ao uso de trapaças, mas na comprovação experimental de que a segurança baseada exclusivamente em *software* é limitada. O isolamento físico do *bot* torna ferramentas como *scanners* de memória e monitores de processos ineficazes contra esta classe de dispositivos.

Como limitações, observa-se que o protótipo depende de coordenadas fixas (ROIs) na interface do usuário. Caso o jogo sofra atualizações visuais significativas, a eficácia do processamento de imagem seria comprometida.

- **Dependência de Resolução/Posição:** O sistema depende de coordenadas fixas (ROIs) e, portanto, exige recalibração manual caso a interface do jogo seja alterada ou se a resolução de captura mude.
- **Lógica de Decisão:** A lógica de estado atual é baseada em regras simples (HP < X, Mana < Y), de modo que o protótipo reage apenas ao estímulo visual imediato e não ao contexto que o personagem está alocado.

Para futuras pesquisas, sugere-se implementar modelos de *Deep Learning* (*YOLO*/*TensorFlow*) para detecção dinâmica de objetos, reduzindo a dependência de coordenadas fixas de interface e desenvolver lógicas de navegação no mapa baseadas em reconhecimento de terreno, extrapolando a cura e mana automáticas.

Conclui-se que a indústria de jogos enfrenta um desafio crescente: a barreira entre o *software* monitorado e o *hardware* invisível. O protótipo TAut serve como um lembrete didático de que a integridade competitiva dependerá, cada vez mais, de abordagens multidimensionais de segurança.

REFERÊNCIAS

YANG, Gefei & ZHANG, Yuqi & LIU, Feihong & GAO, Zishuo. (2023). Machine learning anti-cheating algorithm and a test against computer vision aimbot.

NIE, B.H., Ma, B. (2024). VADNet: Visual-based anti-cheating detection network in FPS games.

MATHEUS, Vinicius & HEINRICH, Tiago & GARCIA, Vinicius & WILL, Newton & OBELHEIRO, Rafael & MAZIERO, Carlos. (2024). O Impacto de Software Anti-cheat na Privacidade do Usuário.

VICENTINE, A. L. .; MENDES DA SILVA, G. C.; DOMINGUES GUEDES DO NASCIMENTO , J. P.; D ALKMIN NEVES , J. E. Software anti-cheat: Uma maneira de prevenir trapaças em jogos multiplayer.

SOUSA, Thiago Castanheira Retes de. Desenvolvimento de um agente inteligente para o MMORPG tibia. 22f. Monografia Graduação, Curso de Ciências da Computação, Universidade Federal do Tocantins, Palmas, 2021.

CIPSOFT. Tibia. Regensburg: CipSoft, 1997. Disponível em: https://www.tibia.com. Acesso em: 10 dez. 2025.

ESPRESSIF SYSTEMS. Products: DevKits. Disponível em: https://www.espressif.com/en/products/devkits. Acesso em: 12 dez. 2025.

ORACLE AMERICA. Java Platform, Standard Edition 21. [S. l.], 2023. Disponível em: https://docs.oracle.com/en/java/javase/21/. Acesso em: 10 dez. 2025.

BYTEDECO. JavaCV: Java interface to OpenCV, FFmpeg, and more. Versão 1.5.10. [S. l.]: Bytedeco, 2024. Disponível em: https://github.com/bytedeco/javacv. Acesso em: 10 dez. 2025.

BRADSKI, Gary; KAEHLER, Adrian. Learning OpenCV: Computer Vision with the OpenCV Library. Sebastopol: O'Reilly Media, 2008.

MICROSOFT. DirectShow. Microsoft Learn, 2024. Disponível em: https://learn.microsoft.com/en-us/windows/win32/directshow/directshow. Acesso em: 10 dez. 2025.

T-VK. ESP32 BLE Keyboard library. Versão 0.3.2. [S. l.], 2025. Disponível em: https://github.com/T-VK/ESP32-BLE-Keyboard. Acesso em: 28 dez. 2025.

BATTLEYE. BattlEye: The Anti-Cheat Gold Standard. [S. l.], 2025. Disponível em: https://www.battleye.com/. Acesso em: 28 dez. 2025.

RIOT GAMES. A Message from the Anti-Cheat Team: Vanguard. [S. l.], 2024. Disponível em: https://support-valorant.riotgames.com/hc/en-us/articles/360046160933-What-is-Vanguard/. Acesso em: 28 dez. 2025.