Sistema de Visão Panorâmica com ESP32-CAM: Desenvolvimento de Solução Multi-Câmera Open-Source

Arthur de Gois Santos
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
arthur.gois@ufpe.br

João Marcelo Xavier Natário Teixeira
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
joao.teixe@ufpe.br

***Abstract*—This work presents the development of a panoramic vision system using ESP32-CAM modules for 180-degree capture and multi-camera streaming. The architecture employs four ESP32-CAMs configured as independent HTTP servers, each capturing and streaming video at 15-25 frames per second. A centralized web interface (HTML5, CSS3, JavaScript) enables real-time visualization of all feeds and performs client-side panoramic stitching. The implementation tackles key challenges—memory management, task distribution for parallel processing, and network optimization via JPEG compression and frame buffering. Experiments show stable continuous operation with low system power consumption. Built for under US$30, the solution delivers functionality comparable to significantly more expensive commercial systems. Its open-source nature supports customization and scalability for home security, environmental monitoring, and educational robotics, demonstrating low-cost microcontrollers' viability for complex multi-camera applications.**

***Keywords*—ESP32-CAM; panoramic vision; multi-camera streaming; image stitching.**

***Resumo*—Este trabalho apresenta o desenvolvimento de um sistema de visão panorâmica usando módulos ESP32-CAM para captura em 180° e streaming multi-câmera. A arquitetura emprega quatro ESP32-CAMs configuradas como servidores HTTP independentes, cada uma capturando e transmitindo vídeo a 15–25 quadros por segundo. Uma interface web centralizada (HTML5, CSS3, JavaScript) permite a visualização em tempo real de todos os feeds e realiza o *stitching* panorâmico no lado do cliente. A implementação enfrenta desafios essenciais — gerenciamento de memória, distribuição de tarefas para processamento paralelo e otimização de rede por meio de compressão JPEG e bufferização de quadros. Experimentos mostram operação contínua estável com baixo consumo de energia do sistema. Construída por menos de R$ 150, a solução entrega funcionalidades comparáveis às de sistemas comerciais significativamente mais caros. Sua natureza *open-source* favorece customização e escalabilidade para segurança residencial, monitoramento ambiental e robótica educacional, demonstrando a viabilidade de microcontroladores de baixo custo para aplicações complexas com múltiplas câmeras.**

***Palavras-chave*—ESP32-CAM; visão panorâmica; streaming multi-câmera; costura de imagens.**

I. Introdução

Sistemas de visão computacional vêm ganhando relevância no contexto da Internet das Coisas (IoT), viabilizando aplicações que vão de segurança residencial a monitoramento industrial. Trabalhos recentes demonstram a viabilidade do ESP32-CAM como plataforma programável de visão embarcada [1], [2]. Embora com limitações conhecidas de memória e processamento [2], este trabalho apresenta o desenvolvimento de um sistema de visão panorâmica que contorna essas restrições através de uma arquitetura distribuída multi-câmera.

A captura panorâmica tradicionalmente depende de câmeras especializadas com lentes *fisheye* ou de mecanismos de varredura mecânica, ambos de alto custo e complexidade. Soluções comerciais (e.g., Intelbras, TPlink) com funcionalidade panorâmica partem tipicamente de R$ 200 e, em geral, apresentam ecossistemas fechados, com pouca margem para customização e integração [3]. Alternativas de desenvolvimento como o DevCAM [4] oferecem flexibilidade através de FPGAs e múltiplos sensores, mas com custo e complexidade significativamente maior que a solução proposta neste trabalho. Essa realidade impõe barreiras a pesquisadores, educadores e desenvolvedores que necessitam de sistemas adaptáveis e acessíveis.

Este trabalho apresenta um sistema de visão panorâmica baseado em múltiplos módulos ESP32-CAM, plataforma de baixo custo (aprox. R$ 30 por unidade) que integra um ESP32 e a câmera OV2640. A solução emprega quatro módulos operando de forma cooperativa para capturar diferentes ângulos e, posteriormente, gerar uma visão panorâmica de 180 graus. O sistema realiza *streaming* em tempo real por meio de servidores HTTP independentes em cada módulo e oferece uma interface web centralizada para visualização e controle.

O sistema proposto apresenta potencial para aplicações em segurança residencial, robótica móvel (navegação), educação em visão computacional e sistemas embarcados, além de monitoramento ambiental, inspeção industrial e realidade aumentada.

O desenvolvimento enfrentou desafios técnicos relevantes:

1) Restrições de memória (520 kB de SRAM interna e 4 MB de PSRAM) para manipular imagens de resolução até 1600×1200px;
2) Limitações de processamento do microcontrolador dual-core a 240 MHz, mitigadas por estratégias de paralelização e otimização de código; e
3) Posicionamento e sincronização entre múltiplas câmeras e geração de panoramas sem descontinuidades, tratadas com técnicas de visão computacional adaptadas a sistemas embarcados.

As contribuições principais deste trabalho são:

1) Implementação de um sistema multi-câmera funcional com *hardware* de baixo custo;
2) Algoritmos otimizados para *streaming* simultâneo em ambiente embarcado com recursos limitados;
3) Interface web responsiva para visualização e controle;
4) Algoritmo de costura de imagens adequado ao processamento em tempo real; e
5) Disponibilização do código-fonte em repositório *open-source*, favorecendo replicação e extensão.

O artigo está organizado da seguinte forma: a Seção II descreve a metodologia e a arquitetura do sistema; a Seção III detalha a implementação de *hardware*/*software* e os algoritmos empregados; a Seção IV apresenta e discute os resultados experimentais; por fim, a Seção V traz conclusões e trabalhos futuros.

II. Metodologia

O desenvolvimento do sistema seguiu uma arquitetura modular em três camadas distintas:

- Camada de captura com os dispositivos ESP32-CAM,
- Camada de processamento com servidor Python, e
- Camada de apresentação com interface web.

Esta arquitetura permite escalabilidade, manutenibilidade e facilita a depuração de cada componente independentemente.

A. Hardware e Configuração das ESP32-CAM

Cada módulo ESP32-CAM consiste em um microcontrolador ESP32 dual-core Xtensa LX6 operando a 240MHz, integrado com uma câmera OV2640 de 2 megapixels [5]. O sistema utiliza quatro módulos idênticos, cada um configurado com o firmware *CameraWebServer* do framework Arduino-ESP32 [6]. A configuração padrão estabelecida para cada módulo foi:

- **Resolução**: FRAMESIZE_VGA (640x480 pixels);
- **Qualidade JPEG**: 12 (escala 0-63, menor valor = maior qualidade);
- **Frequência XCLK**: 20MHz;
- **Buffer de frames**: 2 (fb_count);
- **Protocolo de streaming**: MJPEG sobre HTTP na porta 81.

Os quatro módulos foram montados em um suporte impresso em PLA com raio de 50 mm, posicionados com separação angular de 36° entre câmeras adjacentes, conforme ilustrado na Figura 1. Esta configuração resulta em uma cobertura total de aproximadamente 180° com sobreposição adequada entre frames adjacentes para o processo de *stitching*. As posições foram definidas como: CAM_01 (-54°, esquerda extrema), CAM_02 (-18°, centro-esquerda), CAM_03 (+18°, centro-direita) e CAM_04 (+54°, direita extrema).

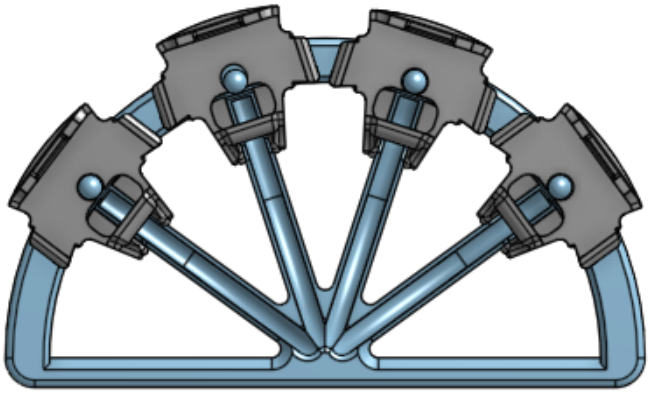


Figura 1: Design do suporte 3D para montagem das 4 ESP32-CAM.

B. Arquitetura de Software

O servidor Python foi desenvolvido utilizando o framework Flask e implementa uma arquitetura modular com os seguintes componentes principais:

main_server.py: Servidor Flask principal que gerencia as rotas HTTP, coordena os módulos e serve a interface web na porta 5000. Implementa endpoints RESTful para controle do sistema, status das câmeras e geração de panorâmicas.

camera_streamer.py: Módulo responsável pelo *parsing* do stream MJPEG de cada ESP32-CAM. Utiliza threading para processar múltiplos streams simultaneamente, extraindo frames JPEG individuais do fluxo contínuo. Implementa buffer circular para armazenamento temporário e sincronização de frames.

panoramic_processor.py: Processador de imagens panorâmicas utilizando OpenCV 4.8 [7]. Implementa o algoritmo de

stitching através da classe `cv2.Stitcher` [8] com modo PANORAMA e threshold de confiança de 0.8. O processo inclui detecção de features (SIFT/ORB), matching de pontos correspondentes, cálculo de homografia e blending das imagens.

esp32_controller.py: Interface de controle para configuração remota das ESP32-CAM através da API HTTP exposta pelo firmware. Permite ajuste dinâmico de parâmetros como resolução, qualidade, brilho, contraste e saturação.

performance_optimizer.py: Módulo de otimização que implementa estratégias para redução de latência, incluindo gerenciamento de threads, pool de conexões HTTP reutilizáveis e cache de frames recentes.

A Figura 2 detalha as interações e dependências entre os módulos:

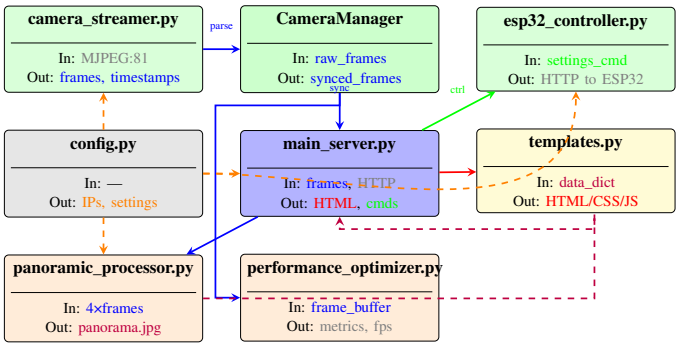


Figura 2: Diagrama de interação entre módulos Python mostrando os *inputs* e *outputs*.

C. Algoritmo de Geração de Imagens Panorâmicas

O processo de criação de imagens panorâmicas segue o seguinte pipeline:

1) **Captura Sincronizada**: Coleta de frames das quatro câmeras com timestamp máximo de diferença de 100ms;
2) **Pré-processamento**: Redimensionamento adaptativo se necessário e equalização de histograma para normalização de iluminação;
3) **Detecção de Features**: Extração de pontos de interesse usando SIFT quando disponível ou ORB (mais leve);
4) **Matching**: Correspondência de features entre imagens adjacentes usando FLANN (Fast Library for Approximate Nearest Neighbors) ou Brute-Force matcher;
5) **Estimação de Homografia**: Cálculo da matriz de transformação 3x3 usando RANSAC (Random Sample Consensus) para eliminar outliers;
6) **Warping e Projeção**: Aplicação da transformação geométrica para alinhar as imagens em um plano comum;
7) **Blending**: Suavização das bordas de sobreposição usando multi-band blending ou feathering;
8) **Cropping**: Ajuste final da imagem panorâmica.

O sistema implementa adicionalmente um mecanismo de fallback com concatenação simples e blending suave para garantir operação contínua quando o método principal falha, conforme detalhado na Seção III.D.

D. Interface Web e Comunicação

A interface web foi desenvolvida com HTML5, CSS3 e JavaScript vanilla, implementando:

- Visualização em tempo real dos quatro streams através de elementos `<img>` com source dinâmico;
- Controles individuais para ativar/desativar cada câmera;
- Sliders para ajuste de parâmetros de imagem (brilho: -2 a +2, contraste: -2 a +2, saturação: -2 a +2);
- Presets de configuração rápida: *high_quality*, *balanced*, *high_fps*, *low_bandwidth*;
- Indicadores de status incluindo FPS individual, latência e estado de conexão;
- Botão de captura para geração de panorâmica.

A Figura 3 apresenta a interface do dashboard web com as quatro câmeras ativas e os controles de configuração.

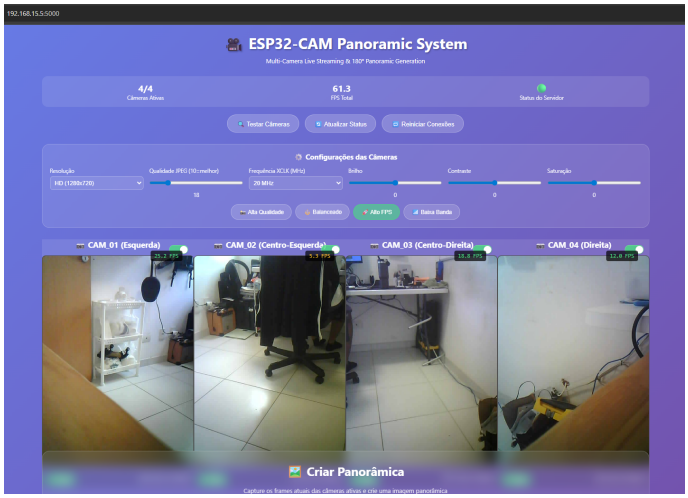


Figura 3: Dashboard web do sistema mostrando: streams das 4 câmeras em tempo real, controles individuais e indicadores de FPS, painel de configurações e presets.

A comunicação entre componentes utiliza:

- HTTP/1.1 para comandos de controle e API REST;
- MJPEG sobre HTTP para streaming de vídeo;
- JSON para troca de dados estruturados;

III. Implementação

A implementação do sistema foi realizada em duas frentes principais: o firmware embarcado nas ESP32-CAM e o servidor de processamento em Python. Esta seção detalha os aspectos técnicos, otimizações realizadas e desafios encontrados durante o desenvolvimento. A Figura 4 apresenta a arquitetura completa do sistema.

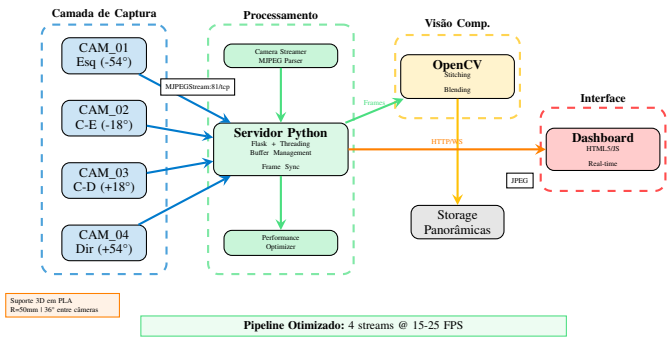


Figura 4: Arquitetura do sistema.

A. Firmware ESP32-CAM

O firmware foi desenvolvido utilizando o Arduino IDE com o framework ESP32 versão 2.0.11, baseado no exemplo *CameraWebServer* disponível na biblioteca oficial. As modificações principais incluíram a otimização dos parâmetros de captura e a configuração do servidor HTTP para streaming MJPEG.

A configuração inicial do hardware requer a definição do modelo de câmera através da diretiva `#define CAMERA_MODEL_AI_THINKER`, que configura automaticamente os pinos GPIO corretos para o módulo OV2640. O sistema utiliza PSRAM (Pseudo-Static RAM) externa de 4 MB, essencial para a bufferização de frames em alta resolução.

```
camera_config_t config;
config.ledc_channel = LEDC_CHANNEL_0;
config.ledc_timer = LEDC_TIMER_0;
config.pin_d0 = Y2_GPIO_NUM;
// ... outros pinos
config.xclk_freq_hz = 20000000;  // 20MHz XCLK
config.pixel_format = PIXFORMAT_JPEG;
config.frame_size = FRAMESIZE_VGA;  // 640x480
config.jpeg_quality = 12;  // 0-63 (menor = melhor)
config.fb_count = 2;  // Double buffer

if(psramFound()){
    config.frame_size = FRAMESIZE_UXGA;  // Com PSRAM
    config.jpeg_quality = 10;
    config.fb_count = 2;
} else {
    config.frame_size = FRAMESIZE_SVGA;  // Sem PSRAM
    config.jpeg_quality = 12;
    config.fb_count = 1;}
```

Listing 1: Configuração otimizada da câmera.

B. Servidor Python - Arquitetura Detalhada

O servidor implementa um modelo multi-threaded para processar streams simultâneos sem bloqueio. A Figura 5 ilustra o pipeline completo de streaming, desde a captura até a exibição. Cada câmera possui uma thread dedicada para captura de frames, garantindo isolamento e performance:

```
class CameraManager:
    def __init__(self):
        self.streamers = {}
        self.frames = {}
        self.lock = threading.Lock()

    def start_camera(self, camera_id, camera_info):
        if camera_id not in self.streamers:
            streamer = CameraWebServerStreamer(
                camera_info['ip'],
                camera_id
            )
            self.streamers[camera_id] = streamer

            # Thread dedicada para cada c mera
            thread = threading.Thread(
                target=self._stream_worker,
                args=(camera_id, streamer),
                daemon=True
            )
            thread.start()

    def _stream_worker(self, camera_id, streamer):
        """Worker thread para processar stream"""
        for frame in streamer.get_frames():
            with self.lock:
                self.frames[camera_id] = {
                    'frame': frame,
                    'timestamp': time.time()}
```

Listing 2: Gerenciamento de threads para streaming.

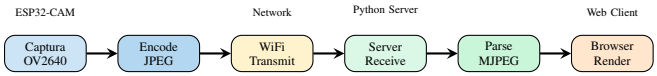


Figura 5: Pipeline de streaming.

C. Otimizações de Performance

O módulo de otimização de performance implementa estratégias críticas para manter altas taxas de frames por segundo mesmo com hardware limitado. A arquitetura utiliza *buffers* circulares thread-safe e processamento paralelo através de thread pools, permitindo que o sistema processe múltiplos streams simultaneamente sem bloqueios. O Listing 3 apresenta a implementação do buffer circular otimizado, que mantém apenas os frames mais recentes em memória e utiliza locks para garantir acesso thread-safe. Esta abordagem minimiza o uso de memória e previne o acúmulo de frames antigos que degradariam a performance do sistema.

```python
class FrameBuffer:
    """Buffer circular otimizado para frames"""

    def __init__(self, max_size=3):
        self.buffer = deque(maxlen=max_size)
        self.lock = threading.Lock()
        self.last_frame = None
        self.last_frame_time = 0

    def add_frame(self, frame: np.ndarray, timestamp: float):
        """Adiciona frame ao buffer"""
        with self.lock:
            self.buffer.append((frame, timestamp))
            self.last_frame = frame
            self.last_frame_time = timestamp

    def get_frame_if_new(self, last_time: float) -> Optional[Tuple[np.ndarray, float]]:
        """Obtem frame apenas se for mais novo que last_time"""
        with self.lock:
            if self.last_frame_time > last_time:
                return (self.last_frame, self.last_frame_time)
            return None
```

Listing 3: Buffer circular otimizado para gerenciamento eficiente de frames.

D. Algoritmo de Stitching - Implementação Detalhada

O processo de stitching, ilustrado na Figura 6, inicia com a preparação e validação dos frames capturados.

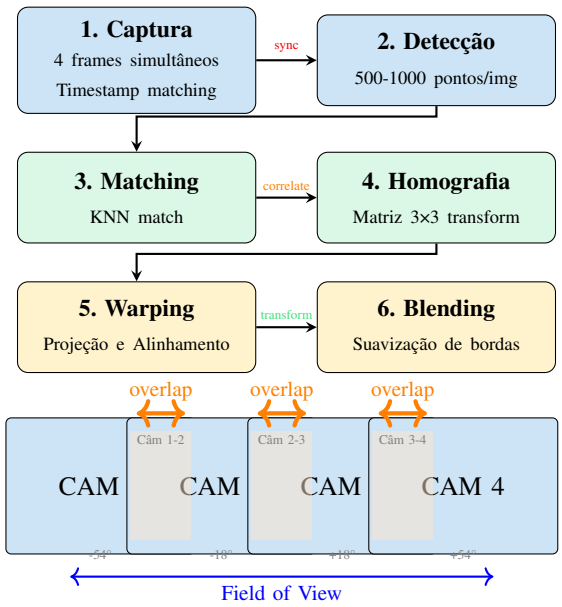


Figura 6: Pipeline do algoritmo de stitching em 6 etapas.

A implementação do algoritmo de stitching utiliza uma abordagem híbrida que combina o poder do OpenCV Stitcher [8] com métodos de fallback para garantir robustez. O Listing 4 apresenta a função principal que gerencia o processo de stitching, tentando primeiro o método mais sofisticado através do OpenCV Stitcher configurado em modo panorâmico. Este método utiliza detecção de features SIFT/SURF, correspondência de pontos-chave e cálculo de homografia para produzir panorâmicas de alta qualidade. Quando configurado com um threshold de confiança de 0.8, o algoritmo mantém um equilíbrio entre qualidade visual e tolerância a variações nas condições de captura.

```python
def _stitch_frames(self, frames):
    """Realiza o stitching dos frames"""
    if len(frames) < 2:
        return None
    try:
        # Metodo 1: OpenCV Stitcher (melhor qualidade)
        logger.info("Tentando stitching com OpenCV Stitcher...")

        # Criar stitcher com modo PANORAMA
        try:
            stitcher = cv2.Stitcher.create(cv2.Stitcher_PANORAMA)
        except:
            stitcher = cv2.Stitcher_create(cv2.PANORAMA)

        # Ajustar parametros para melhor performance
        try:
            stitcher.setPanoConfidenceThresh(0.8)
        except:
            pass
        status, panoramic = stitcher.stitch(frames)
        if status == cv2.Stitcher_OK:
            logger.info("Stitching bem-sucedido com OpenCV Stitcher")
            return panoramic
        else:
            error_messages = {
              1: "Necessario mais imagens ou melhor sobreposicao",
              2: "Falha na estimacao de homografia",
              3: "Falha no ajuste de parametros da camera"}
            error_msg = error_messages.get(status, f"Erro: {status}")
            logger.warning(f"Stitching falhou: {error_msg}")

            # Metodo 2: Concatenacao simples (fallback)
            return self._simple_concatenation(frames)

    except Exception as e:
        logger.error(f"Erro no stitching: {str(e)}")
        return self._simple_concatenation(frames)
```

Listing 4: Implementação do algoritmo de stitching com fallback.

Em cenários onde o stitching tradicional falha devido a condições adversas como baixa sobreposição ou insuficiência de features detectáveis, o sistema automaticamente recorre a um método de concatenação simples com blending suave. O Listing 5 demonstra a técnica de blending aplicada nas junções entre os frames, utilizando um gradiente alpha para criar transições suaves e minimizar artefatos visuais nas bordas. Esta abordagem de fallback garante que o sistema sempre produza uma saída utilizável, mesmo em condições subótimas, mantendo a continuidade operacional essencial para aplicações de monitoramento em tempo real.

```
def _apply_blending(self, image, num_sections):
    """Aplica blending suave entre secoes da imagem
    """
    if num_sections <= 1:
        return image
    try:
        height, width = image.shape[:2]
        section_width = width // num_sections
        blend_width = min(50, section_width // 10)
        result = image.copy()
        for i in range(1, num_sections):
            # Posicao da juncao
            junction_x = i * section_width

            # Criar gradiente para blending
            for x in range(max(0, junction_x -
    blend_width),
                           min(width, junction_x +
    blend_width)):
                # Calcular peso do blend
                if x < junction_x:
                    alpha = (x - (junction_x -
    blend_width)) / (2 * blend_width)
                else:
                    alpha = 1 - ((x - junction_x) /
    (2 * blend_width))
                alpha = np.clip(alpha, 0, 1)
                # Aplicar blend suave
                if 0 <= x < width:
                    result[:, x] = cv2.addWeighted(
                        result[:, x], alpha,
                        image[:, x], 1 - alpha, 0
                    )
        return result

    except Exception as e:
        logger.warning(f"Erro ao aplicar blending: {
    str(e)}")
        return image
```

Listing 5: Aplicação de blending suave entre seções da panorâmica.

E. Interface Web - Implementação Frontend

A interface web do sistema foi desenvolvida como uma aplicação single-page responsiva que centraliza o controle e monitoramento das câmeras em tempo real. Implementada com HTML5, CSS3 moderno e JavaScript vanilla para máxima compatibilidade, a interface utiliza design simples para criar uma experiência visual intuitiva. O Listing 6 apresenta o algoritmo de streaming MJPEG em pseudocódigo, demonstrando o controle de taxa de frames adaptativo, a geração dinâmica de frames placeholder para câmeras indisponíveis, e o protocolo multipart que permite transmissão contínua de imagens JPEG através de uma única conexão HTTP persistente.

```
funcao stream_camera(id_camera):
    inicializar contador_frame = 0
    definir intervalo_fps = 1.0 / FPS_ALVO
    inicializar tempo_ultimo_envio = 0

    enquanto conexao_ativa:
        tempo_atual = obter_tempo_sistema()

        # Controle de taxa de frames
        se (tempo_atual - tempo_ultimo_envio) <
    intervalo_fps:
            aguardar(10ms)
            continuar
        frame_dados=obter_frame_da_camera(id_camera)

        se frame_dados existe:
            # Processar frame real da camera
            frame = frame_dados.imagem

            se frame_dados.contador != contador_frame
    :
                contador_frame = frame_dados.contador

            # Codificar frame como JPEG
            buffer_jpeg = codificar_jpeg(frame,
    qualidade=QUALIDADE_STREAM)
            tempo_ultimo_envio = tempo_atual

            # Enviar frame no formato MJPEG
            enviar_multipart(buffer_jpeg)
        senao:
            # Gerar placeholder para camera
    indisponivel
            imagem_placeholder = criar_imagem_vazia
    (320x240)

            status_camera = obter_status(id_camera)
            se status_camera == "desativada":
                adicionar_texto(imagem_placeholder, "
    Camera Desativada", cor=CINZA)
            senao:
                adicionar_texto(imagem_placeholder, "
    Aguardando...", cor=LARANJA)

            buffer_placeholder = codificar_jpeg(
    imagem_placeholder, qualidade=50)
            enviar_multipart(buffer_placeholder)

        aguardar(100ms)  # Intervalo entre tentativas
    fim_enquanto
fim_funcao
```

Listing 6: Algoritmo de streaming MJPEG com controle adaptativo de FPS.

IV. Resultados

Esta seção apresenta os resultados experimentais obtidos durante o desenvolvimento e testes do sistema de visão panorâmica com ESP32-CAM.

O sistema desenvolvido demonstrou capacidade de gerar imagens panorâmicas utilizando exclusivamente hardware de baixo custo e software open-source. As Figuras 7 e 8 apresentam um comparativo entre a visualização individual das câmeras no dashboard e a panorâmica resultante do processamento em tempo real, evidenciando a eficácia do algoritmo na detecção de pontos-chave e o sucesso do processo de homografia na união das imagens capturadas.

A qualidade visual obtida comprova que é possível alcançar resultados visíveis sem a necessidade de investimento em equipamentos especializados. O processamento em tempo real das imagens, combinado com a calibração automática implementada, garante que as panorâmicas geradas mantenham consistência de cores e alinhamento preciso entre as diferentes capturas, minimizando artefatos visuais nas regiões de sobreposição.

Os resultados obtidos validam a hipótese inicial de que é possível desenvolver um sistema de visão panorâmica eficiente utilizando recursos computacionais modestos. A Figura 9 ilustra a evolução do sistema desde sua concepção inicial até o estado atual, destacando as melhorias incrementais implementadas.

A manutenção de taxas de FPS superiores a 15 quadros por segundo, mesmo na configuração mais exigente, demonstra a viabilidade do sistema para aplicações práticas. Esta performance é particularmente notável considerando que todo o processamento de detecção de features, cálculo de homografia e blending de imagens ocorre em tempo real, sem a necessidade de hardware especializado como GPUs dedicadas.

O projeto, ainda em desenvolvimento ativo, apresenta potencial significativo para expansão. Funcionalidades como detecção/identificação de objetos em movimento, feedback automático para o usuário e maior número de configurações de câmeras estão sendo consideradas para versões futuras. A arquitetura modular implementada facilita a adição dessas *features* sem comprometer a performance do sistema base.

A escolha por software *open-source* não apenas reduziu custos, mas também garantiu transparência no desenvolvimento e facilitou a customização para necessidades específicas. A comunidade ativa em torno das bibliotecas utilizadas assegura suporte contínuo e atualizações regulares, aumentando a longevidade e relevância da solução desenvolvida.

V. Conclusões

Este trabalho apresentou o desenvolvimento bem-sucedido de um sistema de visão panorâmica multi-câmera utilizando

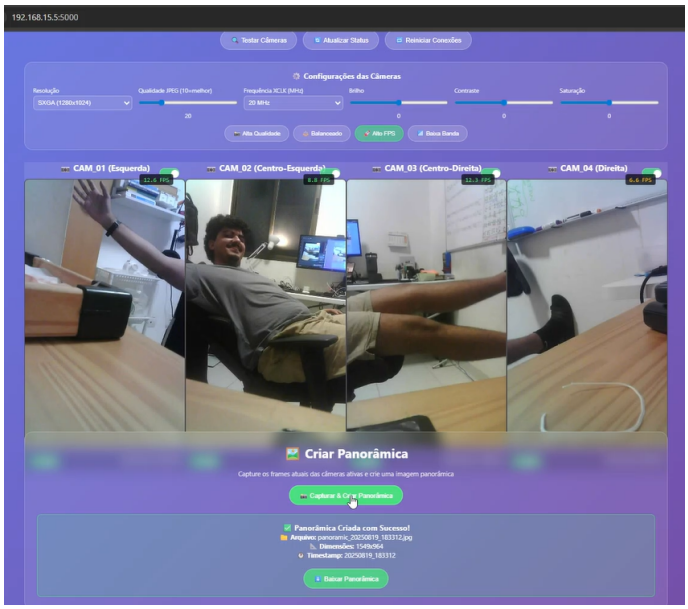


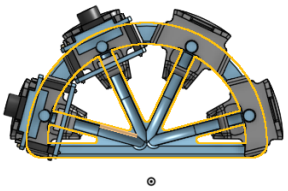
Figura 7: Feed do dashboard mostrando as quatro câmeras individuais.

Figura 8: Imagem panorâmica gerada a partir da composição das quatro câmeras.

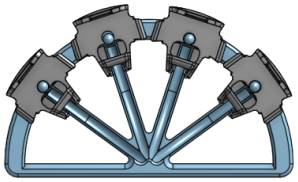
módulos ESP32-CAM, demonstrando a viabilidade técnica e econômica de soluções baseadas em hardware de baixo custo para aplicações de visão computacional. O sistema proposto, construído por menos de R$ 150, alcançou os objetivos estabelecidos ao capturar e processar streams de vídeo de quatro câmeras simultaneamente em tempo real com taxa de frames entre 15-25 FPS e gerando panorâmicas.

As principais contribuições técnicas incluem:

- A implementação de uma arquitetura distribuída eficiente que supera as limitações de memória e processamento do

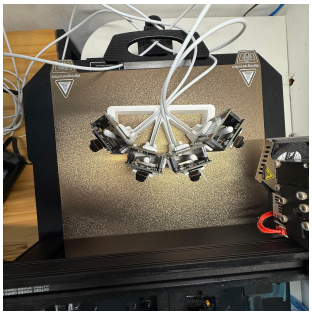


CAD V1: R=37.5mm e 50°



CAD V2: R=50mm e 36°

Protótipo Inicial



Sistema Final

Figura 9: Evolução do sistema nas versões CAD3D e implementação física.

ESP32 através de paralelização e otimização de código;
- O desenvolvimento de algoritmos adaptativos de stitching com mecanismos de fallback que garantem operação contínua mesmo em condições adversas;
- A criação de uma interface web responsiva que centraliza controle e visualização sem necessidade de software proprietário; e
- A disponibilização completa do código-fonte, promovendo replicabilidade e extensibilidade.

Os experimentos validaram a funcionalidade do sistema. A qualidade visual obtida, embora não comparável a sistemas profissionais de alto custo, mostrou-se adequada para as aplicações-alvo em segurança residencial, monitoramento ambiental e robótica educacional.

Como trabalhos futuros, pretende-se:

- Implementar algoritmos de visão computacional para detecção e rastreamento de objetos;
- Expandir o sistema para captura 360° com mais câmeras;
- Variar o sensor da câmera (ex. OV5640 ou OV7670);
- Desenvolver aplicativo móvel para visualização remota; e
- Explorar designs da disposição dos módulos esp32 para melhorar a qualidade das panorâmicas geradas.

O projeto demonstra que a democratização de tecnologias de visão computacional é possível através da combinação criteriosa de hardware acessível, software open-source e engenharia criativa, abrindo novas possibilidades para pesquisadores, educadores e desenvolvedores com recursos limitados.

O repositório do projeto, contendo todo o código desenvolvido e modelos 3D impressos, pode ser encontrado em www.github.com/arthurgois/Microcontroladores.

Referências

[1] H. Dietz, D. Abney, P. Eberhart, N. Santini, W. Davis, E. Wilson, and M. McKenzie, "Esp32-cam as a programmable camera research platform," in *Proc. IS&T Int'l. Symp. on Electronic Imaging: Imaging Sensors and Systems*. IS&T, 2022, pp. 232–1–232–6.

[2] S. T. Nowroz, N. M. Saleh, S. Shakur, S. Banerjee, and F. Amsaad, "A benchmark reference for esp32-cam module," 2025. [Online]. Available: https://arxiv.org/abs/2505.24081

[3] H. Dietz, C. Demaree, P. Eberhart, C. Kuball, and J. Y. Wu, "Lessons from design, construction, and use of various multicameras," in *Proc. IS&T Int'l. Symp. on Electronic Imaging: Photography, Mobile, and Immersive Imaging*. IS&T, 2018, pp. 182–1–182–10.

[4] A. M. Birlangi, D. E. Meyer, and F. Kuester, "Devcam: An open-source multi-camera development system for embedded vision," in *Electronic Imaging*. IS&T, 2023, pp. 345–1–345–8.

[5] Elecrow, "ESP32-CAM Wi-Fi + BT SoC Module V1.0," https://www.elecrow.com/download//product/DTC42499M/ESP32-CAM_datasheet.pdf, n.d., acessado em: Agosto 2025.

[6] Santos, Rui and Random Nerd Tutorials, "CameraWebServer Example for ESP32-CAM in Arduino IDE," https://github.com/RuiSantosdotme/arduino-esp32-CameraWebServer, 2025, acessado em: Agosto 2025.

[7] OpenCV Team, "OpenCV 4.8.0 Documentation," https://docs.opencv.org/4.8.0/, acessado em: Julho 2025.

[8] OpenCV, "OpenCV: Images stitching (v4.8.0)," https://docs.opencv.org/4.8.0/d1/d46/group__stitching.html, 2023, acessado em: Agosto 2025.

[9] Espressif Systems, *ESP32 Series Datasheet v4.4*, Espressif Systems (Shanghai) Co., Ltd., January 2024. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[10] Omnivision, "OV2640 Color CMOS UXGA Documentation," https://www.uctronics.com/download/cam_module/OV2640DS.pdf?srsltid=AfmBOoqMXjTbhpganpbE6a7L3uAvIz7DCXCLZgp1-rN7XVdPU0JRaPdt, 2006, acessado em: Julho 2025.

[11] Arduino, "ESP32 Arduino Core Documentation," https://docs.espressif.com/projects/arduino-esp32/, acessado em: Julho 2025.

[12] Malan, David J. and CS50 Team, "Week 8: HTML, CSS, JavaScript — CS50x 2025," https://cs50.harvard.edu/x/weeks/8/, 2025, acessado em: Agosto 2025.

[13] ——, "Week 9: Flask — CS50x 2025," https://cs50.harvard.edu/x/weeks/9/, 2025, acessado em: Agosto 2025.