UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALFREDO GAUBERT CAPELLA JÚNIOR

Estudo e Implementação de Sistema de Vídeo-Vigilância Inteligente

Trabalho de Conclusão

Prof. Dr. João César Neto
Orientador

Porto Alegre, julho de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Gaubert Capella Júnior, Alfredo

Estudo e Implementação de Sistema de Vídeo-Vigilância Inteligente / Alfredo Gaubert Capella Júnior. – Porto Alegre: PPGC da UFRGS, 2010.

54 f.: il.

Trabalho de Conclusão (mestrado) – Universidade Federal do Rio Grande do Sul. Curso de Engenharia de Computação, Porto Alegre, BR–RS, 2010. Orientador: João César Neto.

1. Vigilância Automatizada. 2. Segmentação. 3. Rastreamento. 4. DaVinci. 5. Visão Computacional. I. Neto, João César. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ABESE	Associação Brasileira das Empresas de Sistemas Eletrônicos de Segurança
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
BSD	Berkeley Software Distribution
CFTV	Circuito Fechado de Televisão
DAG	Directed Acyclic Graph
DSP	Digital Signal Processor/Processing
DVEVM	Digital Video Evaluation Module
DVSDK	Digital Video Software Development Kit
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Arrays
GPP	General Purpose Processor
IP	Intelectual Property
ISA	Instruction Set Architecture
MMU	Memory Management Unit
MoG	Mixture of Gaussians
MPSoC	Multiprocessor System-on-Chip
PC	Personal Computer
PDA	Personal Digital Assistant
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SIMD	Single Instrucion Multiple Data
SoC	System-on-Chip
SRAM	Static Random Access Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit

VLIW	Very Long Instruction Word
xDM	eXpressDSP Digital Media

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O uso de sistemas de circuito fechado de televisão tem crescido enormemente nos últimos anos. Conseqüentemente, cada vez maior a quantidade de informação visual gerada por tais sistemas, tornando impossível o emprego de recursos humanos para monitorar toda esta informação. Desta forma, o desenvolvimento de sistemas de vigilância automatizados é de extrema importância.

Este trabalho visa desenvolver um sistema inteligente de vigilância capaz de detectar e rastrear em tempo real o movimento de pessoas em uma área monitorada por uma câmera. As principais técnicas utilizadas em vigilância automatizada são apresentadas, uma arquitetura para o sistema é proposta e validada em uma plataforma PC e por fim, é desenvolvida uma implementação embarcada deste sistema.

Palavras-chave: Vigilância Automatizada, Segmentação, Rastreamento, DaVinci, Visão Computacional.

Study and Implementation of Intelligent Video Surveillance System

ABSTRACT

The use of closed-circuit television has grown enormously last years. As a consequence of such a growth, more and more visual information is generated for these systems, becoming impossible to employ human resource to monitor all this information. Therefore, it is important to develop automatic surveillance systems.

This project intends to develop an intelligent real-time surveillance system able to detect people and track their movement around an area monitored by a single camera. The main techniques used in automatic video surveillance are presented, then a system architecture is proposed and validated in a PC-based platform and finally, it is developed an embedded implementation for this system.

Keywords: Automatic Surveillance, Computer Vision, DaVinci, Segmentation, Tracking.

1 INTRODUÇÃO

1.1 Circuito Fechado de Televisão

São chamados de circuitos fechados de televisão (CFTV) sistemas que distribuem sinais proveninentes de câmeras, para um determinado ponto de supervisão remoto. Sua principal aplicação é no monitoramento por vídeo de áreas, tais como: bancos, aeroportos e lojas em geral.

1.2 Crescimento do mercado de CFTV

Na última década, o uso de CFTV cresceu enormemente no mundo inteiro, especialmente após os ataques de 11 de setembro. O Reino Unido, país tido como o mais vigiado do mundo, conta com cerca de 4 milhões de câmeras de vigilância. Estima-se que um morador de Londres seja filmado por aproximadamente 300 câmeras de CFTV diariamente. O Brasil não foge à regra. A Abese, com base em número de vendas, estima que o Estado de São Paulo conte com aproximadamente um milhão de câmeras de segurança. Tais câmeras cobrem tanto áreas públicas, como regiões centrais de cidades, quanto áreas privadas, como aeroportos e shopping-centers. Este aumento é estimulado por dois fatores: a queda no preço de sistemas de CFTV e a necessidade crescente por mais segurança. Estudos como Philips [11] e Welsh [17] comprovam redução de alguns tipos crimes em áreas cobertas por câmeras de segurança.

1.3 Necessidade de sistemas de CFTV automatizados

Enquanto alguns sistemas somente gravam as imagens para uso posterior, outros requerem constante monitoramento por parte de um operador a fim de que medidas sejam tomadas no momento que alguma irregularidade é constatada. O número de câmeras que um operador pode efetivamente monitorar também é objeto de pesquisa. Estudos conduzidos pelo Police Scientific Development Branch [16], no Reino Unido, sugerem que é extremamente difícil estabelecer um limite quanto ao número máximo de câmeras que um operador de CFTV pode efetivamente monitorar, uma vez que isto é altamente dependente de parâmetros como complexidade das cenas, tamanho do monitor, número de imagens por monitor, taxa de detecção desejada e competência do operador. Operadores consultados pela pesquisa, no entanto, acreditam que é impossível monitorar efetivamente um número superior a 16 câmeras. Se considerarmos que o custo do hardware destes sistemas tem diminuído continuamente e o custo dos operadores mantém-se relativamente constante, cada vez maior o peso de empregar recursos humanos neste monitoramento. A fim de minimizar tal sobrecarga no custo, tecnologias que auxiliam os operadores de

forma a aumentar sua eficiência vêm sendo desenvolvidas.

1.4 Aplicações Típicas

O interesse em vigilância inteligente e análise da informação visual de vídeos tem crescido muito nos últimos anos, provavelmente graças ao aumento do poder computacional e o sucesso de pesquisas. Além disso, a gama de aplicações da análise de informações visuais é enorme. Por exemplo, Wren [18] propõe um sistema que é capaz de identificar partes individuais do corpo de uma pessoa e seus movimentos. A partir disto é possível criar sistemas capazes de reconhecer gestos e controlar video-games por exemplo. Sistemas de auxílio aos motoristas são outro exemplo de área que vem recebendo grandes investimentos em pesquisas. Informações visuais podem ser usadas para detectar as marcações de uma estrada e emitir um alarme quando o veículo abandona a pista ou invade a pista contrária, evitando que motoristas sonolentos causem acidentes. Já mais próximo à area deste trabalho, a detecção de pedestres e rastreamento de seu movimento tem aplicações em muitas áreas além do contexto deste trabalho. Em shopping-centers pode ser usada para contar número de clientes entrando ou saindo de lojas, identificar os caminhos mais utilizados e as vitrines mais vistas. Estas estatísticas, por exemplo, podem ser extremamente valiosas para a área de marketing de um shopping-center posicionar propagandas.

1.5 Objetivo

Este trabalho possui dois objetivos:

- pesquisar sobre técnicas utilizadas em vigilância automatizada e as várias plataformas voltadas ao mercado embarcado capazes de suportar tais aplicações;
- propôr a implementação de um sistema que adicione algum grau de inteligência à câmera. Tal sistema deve ser compacto o suficiente para ser portado para algum dispositivo embarcado.

1.6 Estrutura do Trabalho

O capítulo 2 do trabalho introduz as técnicas e algoritmos utilizados no contexto de vigilância automatizada. O capítulo 3, então, aborda sistemas de visão computacional dedicados. O capítulo 4 propõe uma arquitetura para o sistema e a valida em um PC, enquanto o capítulo 5 descreve a implementação de parte deste sistema em uma plataforma embarcada. Por fim, o capítulo 6 apresenta as conclusões.

2 SEGMENTAÇÃO E RASTREAMENTO DE OBJETOS

O processamento de toda aplicação baseada em análise de informação visual pode ser visto como um pipeline que elimina informação redundante a cada estágio, como na figura (2.1). A precisão do sistema é, portanto, altamente dependente dos modelos utilizados em cada um destes estágios, e quão capazes eles são de distinguir a informação importante da descartável, considerando o contexto da aplicação.

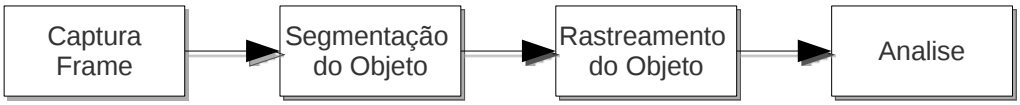

Figura 2.1: Pipeline Genérico de Processamento

Nas próximas seções serão abordadas as técnicas mais relevantes utilizadas neste pipeline.

2.1 Segmentação da Imagem

Segmentação, no contexto de visão computacional, refere-se ao processo de dividir uma imagem em múltiplos segmentos que compartilhem alguma informação visual. É utilizada, tipicamente, para localizar os objetos de interesse na cena, criando uma nova representação da imagem que facilita uma etapa posterior de análise. Existem diversas técnicas e algoritmos endereçados a este problema, baseando-se tanto em dados espaciais quanto temporais, porém, não existe uma solução geral. Desta modo, o conhecimento prévio do domínio da aplicação é de extrema importância.

Especificamente neste trabalho, o objetivo da segmentação é focar a atenção somente nas áreas da imagem em que há movimento. Duas técnicas se sobressaem neste contexto: *background subtraction* e *fluxo ótico*.

2.1.1 Background Subtraction

Esta é uma das técnicas de segmentação mais antigas e mais utilizadas quando a câmera não apresenta movimento em relação aos objetos de interesse. A principal explicação para isto é sua eficácia a um custo computacional relativamente baixo. Consiste em capturar um frame e compará-lo com um frame de referência, comumente chamado de *background*. O resultado é uma imagem binária, com valor '1' nas posições dos pixels que forem considerados diferentes na imagem capturada e de referência.

A principal dificuldade associada a esta técnica não está na obtenção do resultado em si, mas na manutenção do modelo do background. Segundo Toyama [15], existem dez

Figura 2.2: Exemplo de Background Subtraction

problemas canônicos que um modelo de background deve ser capaz de evitar:

Objetos movidos: Um objeto pertencente ao background pode ser movido, porém, ele não deve ser considerado parte do foreground para sempre.

Variações graduais de luz: Variações da luz natural ao longo do dia, por exemplo, podem alterar a aparência do background.

Variações repentinas de luz: Ligar ou desligar uma lâmpada pode alterar a aparência do background. Apesar da similaridade com o item anterior, técnicas bem distintas são aplicadas para combatê-los.

Camuflagem: Problema causado por objetos com coloração similar ao background. Isto pode fazer com que o algoritmo não seja capaz de diferenciá-los.

Distratores: Objetos que devem ser considerados background, porém não são completamente estáticos, como folhas de uma árvore balançando devido ao vento, podem ser erroneamente classificados.

Bootstrapping: Alguns algoritmos requerem um período de treinamento livre de objetos que devam ser considerados foreground. Porém, isto não é possível em alguns ambientes.

Sombras: Sombras de objetos são muitas vezes segmentadas incorretamente, pois além de apresentar movimento, apresentam coloração diferente do background. Podem ser interpretadas como objetos inexistentes ou alterar a forma de objetos corretamente segmentados.

Foreground aperture: Ocorre devido ao movimento lento de objetos uniformemente coloridos. Os valores dos pixels centrais destes objetos não apresentam mudança significativa em frames consecutivos, portanto, nenhum movimento aparente.

Sleeping person: Objetos podem permanecer parados por algum tempo sem ser incorporados ao modelo do background.

Waking person: Quando um objeto inicialmente parte do background move-se, a área do background revelada podem ser, erroneamente, interpretada como um objeto.

Na sua forma mais simples, um primeiro frame livre de objetos é capturado e mantido como frame de referência. Após isto, para cada novo frame capturado, a sua diferença absoluta para o frame de referência é calculada. Se para um dado pixel, esta diferença for superior a um determinado limiar ele é marcado como foreground. Claramente, este método carece de qualquer tipo de adaptação e sofre de quase todos os problemas canônicos mencionados por Toyama.

Wren [18] propôs modelar o background como uma distribuição gaussiana. A média é calculada de acordo com a equação abaixo:

$$\mu_t(x) = \alpha \cdot I_t(x) + (1 - \alpha) \cdot \mu_{t-1}(x) \tag{2.1}$$

onde μ_t é a nova média, I_t é o frame capturado, μ_{t-1} a média anterior e α um parâmetro empírico que busca o equilíbrio entre rápida atualização e estabilidade do modelo. A variância é calculada de maneira similar e um pixel é marcado como foreground caso satisfaça a inequação abaixo:

$$|I_t(x) - \mu_t(x)| > k \cdot \sigma_t \tag{2.2}$$

A utilização de um limiar baseado na variância como critério de decisão minimiza o problema dos distratores, visto que estas regiões devem apresentar maior variância e, consequentemente, um limiar superior. Uma melhora neste algoritmo pode ser feita incluindo uma ideia simples proposta por Koller [5]. Ele propõe que utilize-se um α diferente para pixels classificados como foreground e background na equação (2.1). Por exemplo:

$$\alpha = \begin{cases} 0.1 & \text{se pixel classificado como background} \\ 0.01 & \text{se pixel classificado como foreground} \end{cases}$$

Desta forma, áreas identificadas como foreground são atualizadas mais lentamente do que áreas classificadas como background, e consequentemente problemas como *objetos movidos* e *sleeping person* são atenuados.

O algoritmos vistos até agora modelam o background com uma única média e variância para cada pixel da imagem. Um exemplo típico em que este modelo falha é uma camera externa filmando uma área que apresenta árvores cobrindo parcialmente um prédio. A intensidade do valor de um determinado pixel desta imagem pode representar por vezes uma folha da árvore, o galho da árvore ou a parede do prédio que a árvore cobre. Este pixel, apesar da variação, deveria ser sempre considerado parte do background e uma única distribuição gaussiana não é capaz de modelá-lo. Stauffer e Grimson [14] propuseram um método que é conhecido como *Mistura de Gaussianas (MoG)* para modelar este tipo de comportamento nos pixels. Ele consiste em modelar cada pixel como um conjunto

de K distribuições gaussianas de forma que a probabilidade de se observar um certo valor no pixel x, no tempo t seja:

$$P(x_t) = \sum_{i=1}^{K} \omega_{i,t} \cdot \eta(x_t, \mu_{i,t}, \Sigma_{i,t}) \tag{2.3}$$

onde K é o número de distribuições, $\omega_{i,t}$ o peso da i^{esima} gaussiana no tempo t, $\mu_{i,t}$ a média da i^{esima} gaussiana da mistura no tempo t, $\Sigma_{i,t}$ a matriz de covariância da i^{esima} gaussiana no tempo t e η a função densidade de probabilidade para uma distribuição gaussiana. Cada pixel x_t é comparado com as K distribuições gaussianas até que alguma distribuição *case*[1] com o valor do pixel. É dito que uma distribuição *casa* com o valor do pixel caso no intervalo de até 2.5 desvios padrões da distribuição. Caso o pixel não *case* com nenhuma distribuição, a distribuição menos provável (menor ω) é substituída por uma distribuição com média igual ao valor do pixel corrente, alta variância e baixo peso (ω). Desta forma, o algoritmo é capaz de lidar com variações repentinas de iluminação, por exemplo. Além disso, os parâmetros ($\omega_{i,t}, \mu_{i,t}, \sigma_{i,t}$) são atualizados de maneira similar a equação (2.1) somente para a distribuição que tiver *casado* com o valor do pixel. Este modelo mostrou ser extremamente eficaz e é a base para outros modelos mais sofisticados.

Alguns autores defendem que outros parâmetros, além da média utilizada nos modelos anteriores, apresentam melhores resultados. Cucchiara [2] utiliza a mediana dos últimos n frames como modelo de background. A principal desvantagem deste modelo é que a mediana necessita de um buffer com o valor do pixel nos ultimos n frames para ser calculada. Este alto custo de memória pode impossibilitar que este tipo de algoritmo seja utilizado em algumas aplicações. Em W^4 [4], o autor grava a intensidade mínima(M), a intensidade máxima(N) e a maior diferença absoluta interframe (D) para cada pixel. Um pixel x é então classificado como foreground caso satisfaça a equação:

$$|M(x) - I_t(x)| > D(x) \quad \vee \quad |N(x) - I_t(x)| > D(x) \tag{2.4}$$

Os parâmetros (M, N, D) são inicializados durante uma fase de treinamento livre de objetos que devam ser considerados foreground e atualizados com intervalo de alguns segundos. Novamente, somente os pixels classificados como background são atualizados.

2.1.2 Fluxo Ótico

Fluxo Ótico é a segunda técnica mais utilizada a fim de segmentar objetos em movimento em sequências de vídeo. Esta técnica consiste em estabelecer correspondência entre pequenos blocos de pixels de dois frames consecutivos para aferir seu movimento. O resultado do fluxo ótico é um campo vetorial, cujos vetores representam o deslocamento de cada bloco de pixels.

Baseia-se no fato de que objetos que aparecem na cena no tempo t geralmente aparecem novamente no próximo frame, capturado em $t + \delta t$, deslocados de δx e δy. Isto pode ser reescrito como:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \tag{2.5}$$

onde $I(x, y, t)$ é a intensidade do pixel da posição x, y no tempo t. Assumindo que os

[1] casar usado como tradução para match

deslocamentos entre os frames são pequenos e expandindo a função através da série de Taylor é possível concluir que:

$$\frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial y}\frac{\partial y}{\partial t} = -\frac{\partial I}{\partial t}$$

esta equação é normalmente chamada de *equação de restrição do movimento* e pode ser reescrita em sua forma matricial de maneira mais compacta:

$$\nabla I^T \cdot \overrightarrow{V} = -I_t \tag{2.6}$$

onde I_t representa a derivada parcial $\frac{\partial I}{\partial t}$. A equação 2.6 possui dois termos desconhecidos e, portanto, não é capaz de fornecer uma solução única. Este problema é conhecido como *problema de abertura*. Os diversos métodos propostos para estimar o fluxo ótico resolvem este problema acrescentando equações com restrições adicionais a este sistema.

Sua principal vantagem é a robustez à variações na iluminação. Além disto, provê informação acerca do movimento dos pixels, facilitando etapas posteriores do processamento. Porém, o enorme custo computacional requerido por esta técnica torna proibitivo seu uso em plataformas cujo poder de processamento é limitado.

2.1.3 Filtragem Morfológica

O estágio da segmentação de objetos deve identificar as regiões da imagem que correspondem aos objetos em movimento em um determinado frame capturado. A saída deste estágio é tipicamente uma imagem binária com os pixels marcados como foreground ou background. A filtragem morfológica é um passo extremamente comum ao final desta segmentação. O objetivo desta filtragem é remover pequenos objetos típicamente originados de ruídos. Existe uma vasta gama de operadores morfológicos, porém todos eles originam-se de duas operações básicas: *erosão* e *dilatação*.

A ideia do operador de erosão é percorrer a imagem com uma janela deslizante, de 3x3 pixels por exemplo. O pixel central desta janela recebe o menor valor entre os nove pixels cobertos por ela. O resultado de sua ação é equivalente a computar um mínimo local em uma área da imagem, tornando objetos menores e buracos nos objetos maiores, como mostra a figura 2.3.

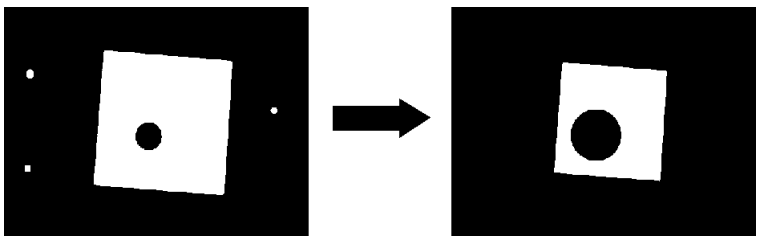

Figura 2.3: Operador de Erosão

A ideia do operador de dilatação é similar, porém o pixel central da janela recebe o maior valor entre os pixels cobertos pela janela. Computando, desta forma, um máximo local da imagem e tornando objetos maiores e buracos menores (Figura 2.4).

No contexto deste trabalho utiliza-se o operador morfológico de *abertura*. A ideia é combinar as duas operações básicas: primeiro uma erosão elimina pequenos blocos

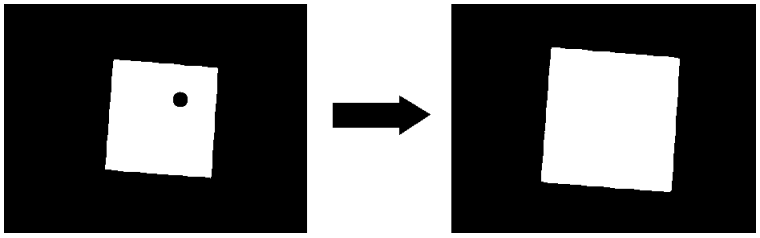

Figura 2.4: Operador de Dilatação

de pixels, geralmente ruídos indesejados e o operador de dilatação preenche pequenos buracos e restaura o tamanho dos objetos (Figura 2.5).

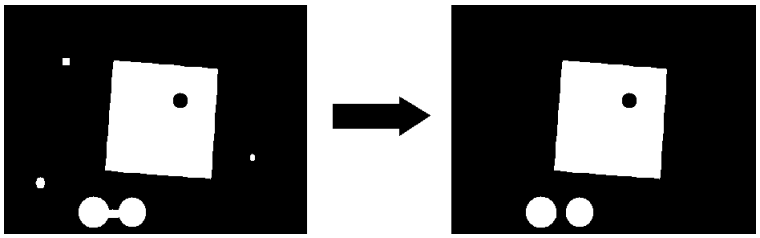

Figura 2.5: Operador de Abertura

2.2 Rastreamento de Objetos

Após o processo de segmentação, descrito na seção anterior, temos uma imagem binária que identifica as regiões da imagem que correspondem aos possíveis objetos em um dado frame. O primeiro passo no rastreamento de objetos é agrupar esses pixels que correspondem aos diferentes objetos e rotulá-los através de algoritmos de *rotulação de componentes conexos*. Além disso, é gerada uma representação de mais alto nível que facilita etapas posteriores do processamento. Uma vez que todos os objetos em um frame foram devidamente identificados, a próxima etapa visa estabelecer correspondência temporal entre os frames. Em outras palavras, identifica os mesmos objetos detectados em diferentes frames ao longo do tempo.

2.2.1 Rotulação de Componentes Conexos

Rotulação de componentes conexos é o procedimento de atribuição de um label único a grupos de pixels, baseado em algum critério de conectividade. Essa extração e rotulação é extremamente comum a muitas aplicações de análise de imagem. Basicamente, os algoritmos varrem a imagem, pixel a pixel, identificando com o mesmo rótulo regiões com pixels adjacentes que compartilhem o mesmo valor (1 no caso de uma imagem binária). Se esta varredura for feita da esquerda para a direita e de cima para baixo denomina-se *forward scan*, ao passo que se for feita de baixo para cima e da direita para esquerda, *backward scan*. Existem duas formas de definirmos a conectividade entre pixels: conectividade-de-4 e conectividade-de-8 [19].

Dois pixels estão 4-conectados se são vizinhos horizontais ou verticais e possuem o mesmo valor e estão 8-conectados se além de possuir o mesmo valor são vizinhos horizontais, verticais ou diagonais. Existem vários algoritmos com esta finalidade, podendo ser classificados de acordo com o número de vezes que varrem a imagem em: *multi-pass*,

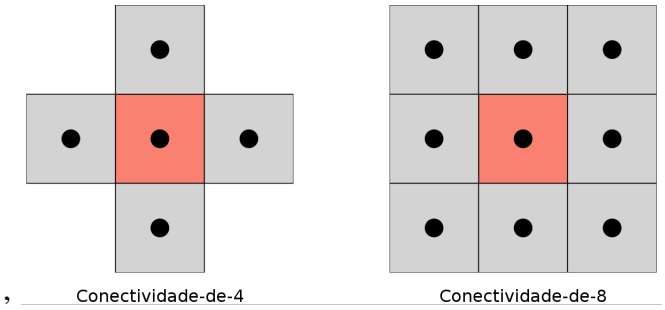

Figura 2.6: Conectividade entre pixels

two-pass e *one-pass* [19].

Os algoritmos mais básicos são os *multi-pass*. Eles varrem a imagem repetidamente, alternando *forward scan* e *backward scan*, propagando rótulos provisórios até que não haja necessidade de mudança nestes rótulos. Algoritmos *two-pass* varrem a imagem duas vezes. Na primeira varredura, atribui rótulos provisórios aos grupos de pixels e armazena equivalências entre os rótulos em alguma estrutura de dados. Na segunda, os rótulos definitivos são atribuídos resolvendo-se as equivalências. Já os algoritmos *one-pass* percorrem a imagem apenas uma vez, porém tipicamente acessam os pixels de forma irregular. Uma vez que acessos sequenciais são muito mais eficazes que acessos randômicos em computadores atuais, nem sempre algoritmos *one-pass* são os mais eficientes. Deste modo, pesquisas na área de otimização destes algoritmos buscam minimizar o número de acessos randômicos à memória.

Além de identificar e agrupar os pixels conexos é gerada nesta etapa uma representação de mais alto nível dos objetos. Tipicamente, uma lista dos objetos contendo informações relevantes para as próximas etapas do processamento, tais como área, coordenadas do centróide, *bounding box* e densidade. É comum também que se descarte desta lista objetos menores que um tamanho pré-determinado. O objetivo disto é evitar que pequenos blocos, provavelmente originados de ruído, sejam propagados no pipeline do processamento.

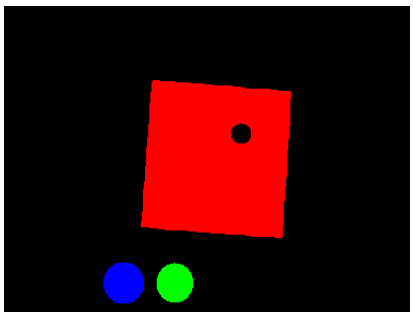

Figura 2.7: Rotulação de Objetos Conexos

2.2.2 Processo de Rastreamento

Uma vez que os objetos de foreground foram devidamente extraídos, eles precisam ser rastreados ao longo dos frames. Esta etapa é extremamente dependente da qualidade da segmentação, porém, idealmente deve ser capaz de lidar com imperfeições no resultado desta. Podemos dividir esta tarefa em duas:

- detectar novos objetos que aparecem no campo de visão da câmera, inicializando devidamente a estrutura necessária para rastreá-los e identificar quando objetos que estão sendo rastreados deixam a cena, liberando as estruturas de dados que os referenciam.

- estabelecer correspondência entre regiões de foreground extraídas de um frame com os objetos sendo rastreados em determinado momento.

Rastrear objetos isolados é relativamente simples, porém este processo pode se tornar bastante complexo na ocorrência de algumas complicações. Um exemplo disso, são pessoas andando próximas o suficiente para serem segmentadas como um único objeto. Na ocorrência deste tipo de complicação as características visuais destes objetos como cores e formas podem se tornar ambíguas, contudo o rastreador deve ser capaz de lidar com isso. Outra dificuldade é a ocorrência de oclusão, ou seja, um objeto não estar visível ou estar somente visível parcialmente durante alguns frames. Além disto tudo, o comportamento humano é imprevisível, de forma que é impossível usar um preditor exato para velocidade ou direção do movimento. Portanto, o modelo utilizado nesta etapa é de fundamental importância.

Existe uma vasta gama de pesquisa nesta área e as soluções propostas são dependentes do domínio da aplicação. Por exemplo, Wren et al. [18] propôs um sistema com a finalidade de tradução automática da linguagem de libras. Para tal, é necessário que o modelo utilizado no rastreamento identifique as partes do corpo e as siga separadamente. Outras aplicações não necessitam de um nível tão alto de detalhamento, como no contexto deste trabalho, cuja finalidade é somente determinar a posição atual da pessoa na cena.

Masoud e Papanikolopoulos [10] desenvolveram um sistema no qual os pedestres são modelados como blocos quadrados com um determinado comportamento dinâmico. O sistema proposto tinha robustez contra oclusões parciais e totais estimando os parametros do pedestre através do método conhecido como filtros de kalman.

Rossi e Bozzoli [12] evitaram o problema da oclusão montando a câmera verticalmente à fim de rastrear e contar pessoas passando em um corredor.

Em W4 [4], os autores primeiro estimam a *bounding box* e o centróide do objeto no tempo t com base em frames anteriores. As bounding boxes de objetos no frame corrente que se sobrepuserem à bounding box estimada e cuja diferença entre os centróides estiver abaixo de um limiar são consideradas candidatas a objeto. Após isto, é computada a correlação binária das bordas entre as silhuetas para encontrar a melhor correspondência.

Fuentes e Velastin [3] o fazem de maneira similar, utilizando informação relativa à sobreposição das bounding boxes como critério para correspondência dos objetos entre os frames, porém sem a etapa da predição. Além disto, propõe que se use a cor ao invés da correlação das bordas como método adicional para resolver eventais conflitos.

Já Latecki e Miezianko [7] localizam os objetos de frames anteriores através de uma função de custo baseada em estatísticas dos objetos extraídas na etapa de *rotulação*, tais como posição e bounding box. A oclusão entre objetos é resolvida estimando sua posição através do cálculo de seu vetor de movimento. Caso não reapareça dentro de um determinado intervalo de tempo, o objeto é incorporado à silhueta mais próxima.

Yang et al [20] caracterizam os objetos utilizando a cor e o histograma de orientação das bordas do objeto. Além disto, lidam com a oclusão utilizando um modelo de predição baseado em filtro de partículas. O filtro de partículas gera múltiplas hipóteses sobre o estado dos objetos no frame $t + 1$ através de um gerador pseudo-randômico. Um estágio

posterior aplica modelos de observação que avaliam a probabilidade de cada uma das destas hipóteses, à fim de atualizar o estado atual do objeto sendo rastreado.

Métodos para rastreamento geralmente tentam estabelecer a relação temporal entre os objetos segmentados através de funções de custo que comparam características da silhueta atual do objeto, com suas características no passado. Oclusões são geralmente tratadas por algum tipo de preditor, baseado no comportamento dinâmico do objeto no passado.

3 VISÃO COMPUTACIONAL EMBARCADA

Em contraste com sistemas computacionais de propósito geral, projetados para serem flexíveis e atender uma grande variedade de aplicações, sistemas embarcados são projetados para se dedicar à tarefas específicas. Sua vantagem está justamente nesta especialização, permitindo que o desenvolvedor dimensione hardware e software da melhor forma possível, atingindo o desempenho desejado com o menor custo e consumo de energia possíveis.

Sistemas embarcados que empregam algum tipo de aplicação da visão computacional têm tornado-se cada vez mais comuns em nosso dia-a-dia, principalmente devido ao advento de dispositivos, tais como: câmeras digitais, celulares e PDAs. O lançamento de novas plataformas voltadas a este mercado torna possível que algoritmos cada vez mais sofisticados sejam implementados e, consequentemente, sistemas cada vez mais complexos sejam criados. Estes novos sistemas impõem novos desafios aos desenvolvedores na busca pelo desenvolvimento de melhores soluções com o menor custo.

Este capítulo procura explorar a metodologia de desenvolvimento, algoritmos e plataformas utilizadas em sistemas dedicados à visão computacional.

3.1 Componentes de Hardware

A maioria das operações relacionadas ao processamento de vídeo são computacionalmente intensivas. Por exemplo, o volume de dados para processar o vídeo de uma única câmera, com resolução e taxa de frames típica (640x480/30fps), chega à 27MB/s. Além disso, a maioria dos algoritmos de baixo nível utilizados neste contexto executam centenas de operações elementares sobre cada um destes pixels.

A variedade de arquiteturas disponíveis para os projetistas de tais aplicações é enorme e variam desde circuitos dedicados como ASICs e FPGAs até processadores comuns e DSPs. Além disso, há plataformas que combinam tais arquiteturas em Systems-on-Chip (SoC).

3.1.1 Field Programmable Gate Array (FPGA)

Um FPGA é um dispositivo semicondutor, no qual a lógica do circuito e suas conexões podem ser modificadas de acordo com as necessidades da aplicação. O *design* da lógica determina qual a sua real funcionalidade. Há três tipos de tecnologias quanto à programação de FPGAs: antifusível, SRAM e FLASH. FPGAs baseados em antifusíveis não são reprogramáveis. SRAMs são reprogramáveis, porém voláteis; ou seja, o chip deve ser reprogramado toda vez que for ligado. FLASH é não volátil, como antifusível, mas possui a vantagem de poder ser reprogramada tantas vezes quanto forem necessárias.

A lógica implementada pelo circuito é, geralmente, especificada através de linguagens de descrição de hardware, tais como Verilog e VHDL. Ferramentas de EDA, então, sintetizam esta descrição e geram o *netlist* do circuito. Etapas de mapeamento tecnológico, roteamento e posicionamento são executadas, normalmente por ferramentas disponibilizadas pelo próprio fabricante do FPGA, e geram um arquivo de programação que deve ser passado ao dispositivo para implementar o circuito desejado.

A arquitetura básica de um FPGA consiste de blocos de lógica configurável (CLBs), pads de I/O e canais de roteamento com interconexões programáveis. Alguns FPGAs modernos combinam estes blocos tradicionais para lógica programável com microprocessadores embarcados e periféricos. Exemplos disto são as séries Virtex-II PRO e Virtex-4 da Xilinx que possuem um ou mais processadores PowerPC no mesmo chip do FPGA. Alternativamente pode-se utilizar em FPGAs os chamados processadores soft-core, disponibilizados por fabricantes como Xilinx e Altera. Neste caso, é comprado o direito ao uso da propriedade intelectual (IP) e o processador é implementado na lógica programável do FPGA. A vantagem dessa abordagem é a flexibilidade, pois estes processadores são reconfiguráveis, tal que suas características podem ser adicionadas ou retiradas de acordo com a necessidade.

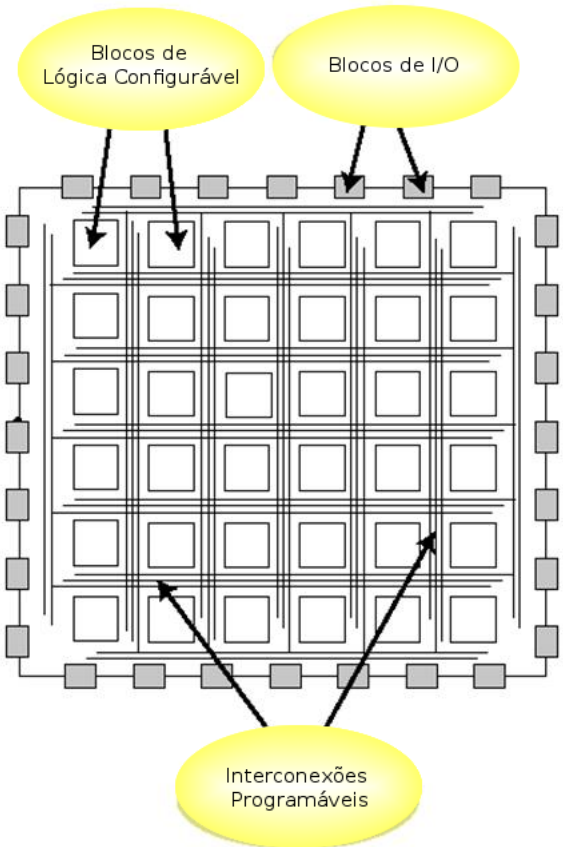

Figura 3.1: Arquitetura Básica do FPGA

O grande benefício do FPGA é a flexibilidade em termos da lógica, oferecendo um enorme paralelismo no fluxo de dados e no processamento. Porém, implementar um algoritmo utilizando eficientemente o paralelismo oferecido nem sempre é uma tarefa trivial. Outras desvantagens de FPGAs são o alto consumo e velocidades do clock inferior a processadores DSP típicos.

3.1.2 Processadores de Sinal Digital (DSP)

Um DSP é similar a um processador de propósito geral (GPP) em muitos aspectos: possui uma lógica fixa entre os gates, possui um conjunto de instruções (ISA) limitado e executa estas instruções sequencialmente. O que o diferencia de processadores com propósito geral é, justamente, que seu projeto leva em conta as operações habituais em processamento de sinais. Portanto, seu conjunto de instruções é otimizado para operações sobre matrizes, fornecendo instruções SIMD (Single Instruction, Multiple Data) especializadas. Alguns exemplos destas instruções são dados na tabela 3.1.2.

Instrução	**Descrição**
AVGU4	Média entre 4 pares de dados
BITC4	Conta bits em '1' em 4 operandos
CMPGT2	Duas comparações (>=) em paralelo
DOTPU4	Produto escalar entre 2 vetores de 4 dimensões
MAXU4	Retorna valor máximo entre 4 operandos
SADDU4	Soma saturada de 4 pares de dados
SUBABS4	Diferença absoluta entre 4 pares de dados

Tabela 3.1: Instruções do Processador TMS320C64x da Texas Instruments

Além disso, muitos DSPs são capazes de executar diversas instruções em paralelo. Este paralelismo é, geralmente, explorado por arquiteturas do tipo VLIW (Very Long Instruction Word). A principal diferença destas arquiteturas para arquiteturas superescalares é o momento em que o paralelismo entre as instruções é descoberto. O compilador é o responsável por descobrir a dependência entre as instruções e escaloná-las para as diferentes unidades funcionais. Toda a complexidade desta tarefa é, portanto repassada a ele, tornando a unidade de controle de um processador VLIW extremamente simples. Esta unidade de controle simples, aliada a um pipeline tipicamente profundo[1], torna *branches* especialmente custosos em DSPs.

Outra característica típica de DSPs é a organização da memória. Eles utilizam arquiteturas conhecidas como *Harvard*, baseadas na separação completa das memórias de instruções e dados. Isto permite que se obtenha melhor desempenho do que arquiteturas de von Neumann, visto que o processador pode buscar instruções e dados, simultaneamente, em barramentos diferentes. Além disto, há uma maior flexibilidade, pois pode-se utilizar tecnologias completamente diferentes para cada memória.

Há diversos DSPs disponíveis comercialmente, inclusive plataformas heterogêneas que incluem processadores de propósito geral no mesmo chip do DSP. Além disto, existem DSPs especializados para processamento de vídeo e imagem, conhecidos como processadores de mídia. Estas plataformas tendem a incluir múltiplas unidades de DMA e I/O streams otimizadas para transferência de pixels.

[1] os processadores da família TMS320C6000 da Texas Instruments possuem pipeline com 11 (ponto fixo) ou 16 (ponto flutuante) estágios

3.1.3 System-on-Chip (SoC)

System-on-Chip é um termo um pouco nebuloso. Ele se refere a circuitos integrados que contém todos os componentes essenciais para um sistema embarcado no mesmo chip. SoCs consistem tipicamente de:

- Um processador comum ou um DSP. Alguns possuem mais de um processador e são chamados *Multiprocessor System-on-Chip* (MPSoC);
- Blocos de memória como ROM, RAM, FLASH;
- Interfaces externas como USB, FireWire, Ethernet, cartões de memória;
- Interfaces analógicas como conversores A/D e D/A;
- Periféricos como timers, watchdogs;
- Hardwares aceleradores específicos.

A maior parte dos SoCs possuem um GPP, tal como ARM, MIPS ou PowerPC como elemento central. Este elemento central é, então, auxiliado por DSPs ou hardwares aceleradores específicos. O que torna estes circuitos verdadeiros *Sistemas em um Chip* é o grande número de periféricos incluídos no mesmo circuito integrado. DSPs modernos, como os citados no final da seção anterior, podem ser classificados também como SoC. Um exemplo conhecido são os processadores DaVinci da *Texas Instruments*. São processadores otimizados para aplicações de vídeo, possuindo um núcleo ARM para tarefas de controle e um DSP, para aceleração de algoritmos de processamento de imagem. Possui, por exemplo, aceleradores específicos para cálculo de histogramas, redimensionamento de imagem, foco automático, balanço de branco, além de controladores de DMA e conversores D/A e A/D.

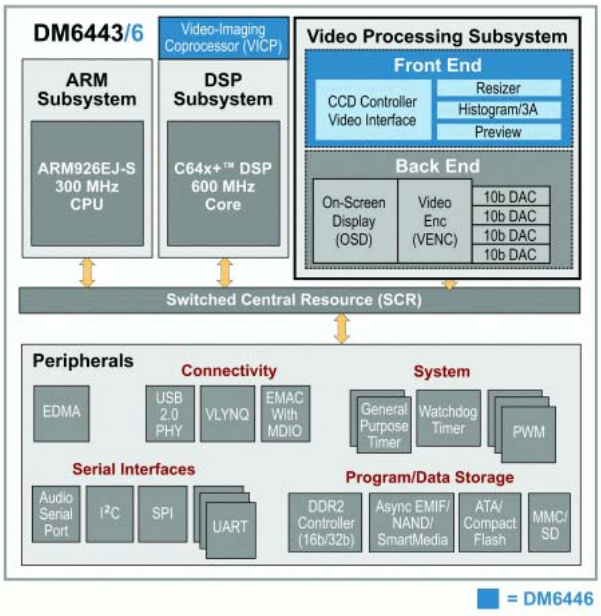

Figura 3.2: Diagrama de Blocos do DaVinci DM644X

3.2 Algoritmos

Sistemas embarcados costumam possuir uma série de restrições, tais como: baixo consumo, baixo custo e quantidade limitada de memória. Por outro lado, processamento de imagem e algoritmos de visão computacional, em geral, requerem grande poder computacional e grandes quantidades de memória para efetuar o processamento em tempo real. Portanto, algoritmos destinados a estes sistemas requerem esforços especiais no design. A performance de um algoritmo em um DSP ou FPGA difere bastante de sua respectiva performance em um processador de uso geral. As seguintes características tornam algoritmos melhores para execução nestas plataformas [6]:

- streams de dados e acessos sequenciais aos dados,
- múltiplas streams de dados independentes,
- altas taxas de dados com poucas instruções por dados,
- tamanhos fixos dos pacotes de dados,
- computações que podem ser quebradas em estágios de pipeline,
- altas taxas de dados com poucas instruções por dados,
- operações que requerem somente valores com precisão de ponto fixo,
- algoritmos paralelizáveis no nível de módulos e instruções.

Kölsch e Butner [6] classificam os algoritmos de acordo com o seu padrão de acesso à memória (Tabela 3.2). Para o melhor desempenho possível, o algoritmo deveria acessar somente um píxel por vez e a sequencia de pixels acessados ser previamente conhecida.

A tabela 3.2 pode ser interpretada da seguinte forma: quanto mais para cima da tabela, menor a interdependência entre os dados e mais adequado o algoritmo para implementação em DSPs e FPGAs. Por outro lado, quanto mais para baixo da tabela, maior a interdepêndencia entre os dados e, consequentemente, mais adequado o algoritmo para implementação em um GPP. Portanto, funções ideais para implementação em DSPs e FPGAs são tipicamente funções de mais baixo nível, que apresentam pouca interdependência.

Interdependência	Exemplos de Algoritmos
Processamento de Pixels: a imagem é percorrida apenas uma vez e o novo valor de um pixel é dependente somente do valor de um píxel lido.	• Lookup-tables • Conversão de espaço de cores • Limiar de intensidade do pixel • Operações aritméticas
N-pass: a imagem é percorrida multiplas vezes e é necessário espaço para armazenamento temporário de dados, porém somente um pixel determina o novo valor do pixel.	• Mínimo, máximo, média • Equalização de histogramas • Transformadas de Hough
Acesso a blocos de tamanho fixo: o valor de saída é determinado pelo valor dos pixels de uma área de tamanho fixo da imagem.	• Filtros • Operadores Morfológicos • Wavelets
Acesso global, independende do dado: o valor de saída é determinado pelo valor de múltiplos pixels de qualquer parte da imagem, porém o padrão do acesso aos pixels é conhecido a priori.	• Viola-Jones • Correção de distorções
Acesso randomico, dependente do dado: o valor de saída é determinado pelo valor de múltiplos pixels de qualquer parte da imagem. Os pixels acessados dependem do valor dos pixels lidos.	• Algoritmos de preenchimento (flood-fill) • Determinação de contornos

Tabela 3.2: Interdependência de dados

3.3 Metodologia de Design

A metodologia de design empregada na implementação de sistemas de visão computacional embarcados é um problema crítico. Isto se deve à complexidade tanto das aplicações, quanto das plataformas alvo. A fim de salientar os aspectos importantes no fluxo de design, Saha [13] divide este problema em cinco sub-problemas inter-relacionados: especificação e modelagem, mapeamento e particionamento, escalonamento, exploração do espaço de design e geração de código.

3.3.1 Especificação e Modelagem

O primeiro passo para uma implementação eficiente é a utilização de um modelo adequado para sua especificação. Existem diversos modelos e linguagens formais especialmente desenvolvidos com este propósito. Um *design* pode ser representado como um conjunto de blocos que interagem entre si e com o ambiente. Os modelos formais são utilizados para definir o comportamento desses blocos. Exemplos destes modelos incluem máquinas de estados (FSM), redes de Petri e fluxo de dados. Linguagens formais, por outro lado, permitem que o projetista especifique as interações entre os componentes e o conjunto de restrições, às quais o sistema está sujeito. Exemplos destas linguagens incluem ML, linguagens de fluxo de dados (e.g., Lucid, Haskell) e linguagens síncronas (e.g., Lustre, Esterel, SCADE). A utilização destes modelos permite uma melhor

compreensão do comportamento do sistema e, consequentemente, antecipar detecção de problemas no design.

3.3.2 Mapeamento e Particionamento

Após o modelo do sistema e a plataforma de implementação estarem definidos, é necessário particionar as tarefas e mapeá-las para as diferentes unidades de processamento disponíveis. Este é um problema de otimização complexo, no qual os objetivos podem ser, por vezes, conflitantes. A simples paralelização das tarefas não garante uma imple-

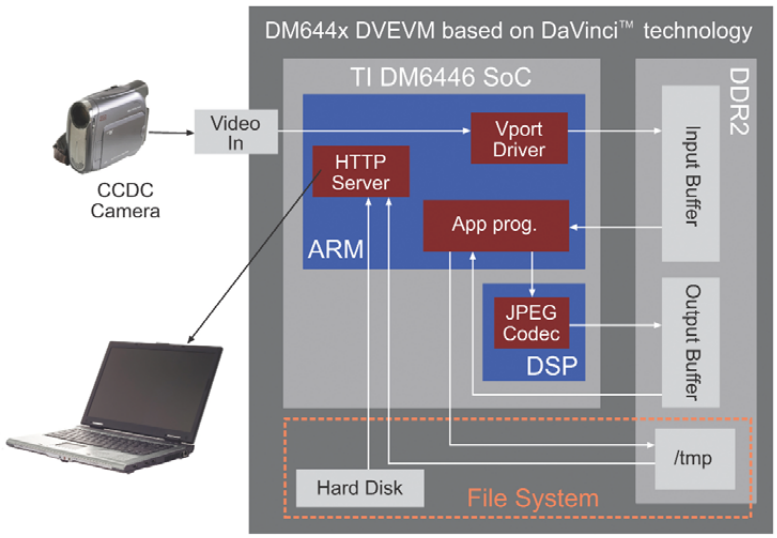

Figura 3.3: Fluxo de dados do software de demonstração Motion JPEG baseado da plataforma DaVinci

mentação eficiente. É importante que se avalie os overheads inerentes à tal paralelização, tais como: comunicação entre unidades de processamento, sincronização e gerenciamento da memória; portanto, a granularidade desta divisão é fator fundamental do design. Em processamento de vídeo, é comum dividir a imagem em blocos e então, processá-los ou transmiti-los. A solução geralmente envolve identificar caminhos críticos para a performance. Desta forma, é importante que os requisitos de desempenho estejam bem definidos para não haver desperdício de recursos.

3.3.3 Escalonamento

Escalonamento refere-se à tarefa de determinar a ordem de execução das várias funções nos diferentes sub-sistemas, tal que os requisitos de performance sejam atingidos. Ele pode ser estático, dinâmico ou uma combinação dos dois. Escalonamento estático é o mais utilizado em sistemas embarcados, pois estes sistemas tendem a ser previsíveis e assim, evita-se o *overhead* associado ao escalonamento dinâmico. Algumas vezes as decisões de escalonamento dependem de entradas ou de resultados intermediários que não podem ser preditos. Por isso, comumente usa-se uma combinação dos dois. Parte do escalonamento é determinado antes da execução e parte em tempo de execução.

Aplicações de processamento digital de sinais (DSP) são comumente descritas por grafos de fluxo de dados.

3.3.4 Exploração do espaço de design

Exploração do espaço de design envolve a avaliação do design do sistema e busca por alternativas em relação a aspectos importantes para a implementação. Na maior parte dos casos, envolve examinar múltiplos designs e escolher aquele que oferece a melhor relação custo-benefício.

Uma ferramenta de exploração do espaço de design eficiente pode ter impactos significativos no custo, performance e consumo do sistema, focando a atenção do projetista nas regiões promissoras do espaço de soluções. Tais ferramentas podem, inclusive, ser usadas em conjunto com as outras tarefas do design.

3.3.5 Geração de código e verificação

Após todos os passos envolvendo a definição das tarefas da aplicação e o seu mapeamento nos recursos de hardware, o próximo passo envolve a geração de código propriamente dita.

O último passo antes do lançamento do produto envolve teste, verificação e validação para garantir que o produto atenderá à especificação.

4 IMPLEMENTAÇÃO PARA PC

Este capítulo descreve a implementação de um sistema para detectar e rastrear pessoas em uma área monitorada por vídeo. O sistema desenvolvido processa as imagens capturadas por uma única câmera que é verticalmente posicionada, a fim de evitar oclusões totais e facilitar o rastreamento.

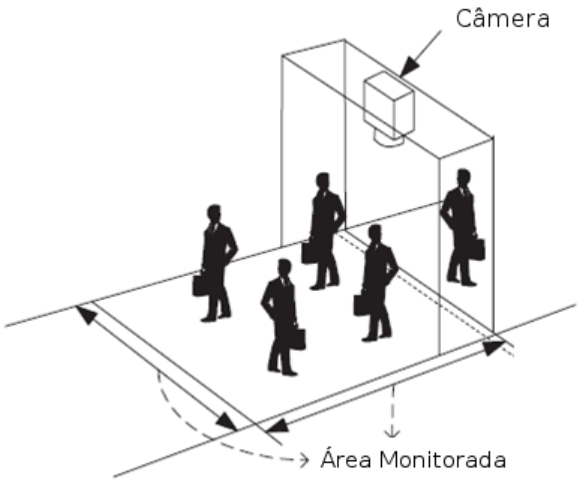

Figura 4.1: Definição do Sistema

As primeiras etapas da implementação envolvem explorar diferentes alternativas para o projeto, identificar algoritmos adequados e avaliar diferentes combinações que possam atender à especificação. Apesar do objetivo final deste trabalho ser um sistema embarcado, não é interessante lidar com as restrições impostas por tais plataformas nesta etapa inicial do desenvolvimento. Por outro lado, computadores pessoais (PCs) oferecem uma enorme flexibilidade, além de possuir uma vasta gama de bibliotecas disponíveis. Desta forma, tornam-se ideais para esta fase inicial do projeto.

Primeiramente é dada uma visão geral da arquitetura implementada. Então, as bibliotecas utilizadas e os módulos são explicados a fundo. Por fim, os resultados obtidos em testes são analisados.

4.1 OpenCV

OpenCV (Open Source Computer Vision) é uma biblioteca, originalmente desenvolvida pela Intel, voltada ao processamento de imagens em tempo real. A biblioteca é escrita em C e C++ e pode ser executada em Linux, Windows e Mac OS. Ela fornece uma infraestrutura simples para desenvolver aplicações de visão computacional.

A biblioteca é dividida em quatro componentes principais, como mostrado na figura 4.2. O componente CV contém os algoritmos básicos de processamento de imagem e algoritmos de visão computacional de alto nível; ML é a biblioteca com funções de aprendizagem de máquina, tais como classificadores estatísticos e *clustering*. HighGUI contém as rotinas de I/O, além de funções para carregar e salvar vídeos. Por fim, CXCore contém os algoritmos e estruturas de dados básicas.

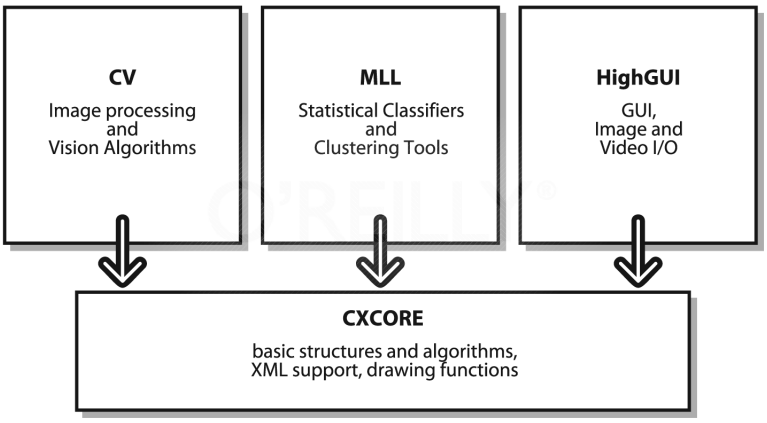

Figura 4.2: Estrutura do OpenCV [1]

OpenCV é uma biblioteca de código aberto e distribuída sob os termos da licença BSD. Esta licença não estabelece limitações quanto ao uso do código, apenas os créditos devem ser mantidos. Desta forma, um produto comercial que utilize esta biblioteca pode ser comercializado sob qualquer outra licença proprietária. Além disso, não há nenhuma necessidade de disponibilizar seu código fonte.

4.2 Arquitetura do Sistema

A arquitetura geral do sistema é mostrada na figura 4.3. Os primeiros passos do sistema são a manutenção do modelo de *background* e a estimativa do *foreground*. Isso resulta em uma imagem binária na qual os pixels considerados em movimento têm o valor '1' e todos os demais '0'. Esta imagem binária passa por uma filtragem morfológica, a fim de eliminar pequenas regiões segmentadas incorretamente. Após isto, os pixels conexos são agrupados e é gerada uma representação de mais alto nível para a etapa de rastreamento. Esta é responsável por localizar os objetos em cada frame e identificar sua trajetória ao longo do tempo. Finalmente, a interação entre os objetos e sua trajetória é analisada a fim de obter estatísticas ou ativar algum alarme.

Os módulos foram desenvolvidos na liguagem C, com o auxílio da biblioteca OpenCV. As próximas subseções detalham a implementação de cada módulo.

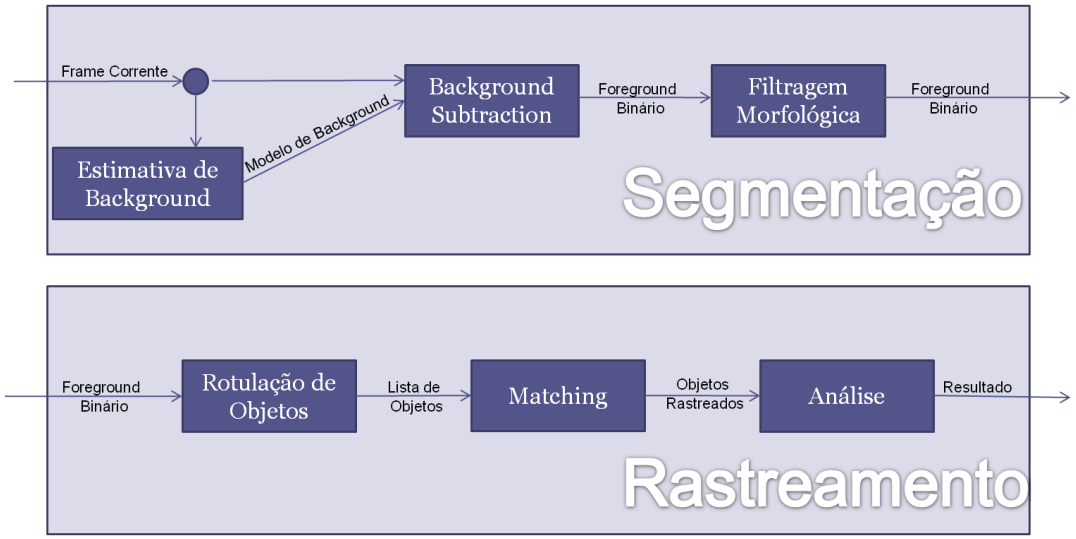

Figura 4.3: Pipeline de processamento proposto

4.2.1 Segmentação

Como mencionado anteriormente, segmentação refere-se ao processo de dividir uma imagem em múltiplos segmentos que compartilhem alguma informação visual. É utilizada, tipicamente, para localizar os objetos de interesse na cena. As principais técnicas e algoritmos endereçados a este problema são abordadas na seção 2.1.

O algoritmo de segmentação implementado neste trabalho é baseado no proposto por Manzanera e Richefeu [9]. Eles propuseram modelar o *background* através de uma aproximação recursiva da mediana da intensidade dos pixels que compõem a cena. Tal aproximação é baseada na modulação Σ-Δ, bastante utilizada em conversores A/D. A ideia é utilizar somente operações elementares de comparação e incremento/decremento para estimar a mediana.

Inicialização
para cada pixel x
se $M_0(x) = I_0(x)$

Para cada frame capturado
para cada pixel x
se $M_{t-1}(x) < I_t(x)$ então $M_t(x) = M_{t-1}(x) + 1$
se $M_{t-1}(x) > I_t(x)$ então $M_t(x) = M_{t-1}(x) - 1$

Tabela 4.1: Mediana Σ-Δ

O cálculo da estimativa da mediana é apresentado na tabela 4.1, onde M_t representa o valor da mediana e I_t o frame corrente no tempo t. Ao contrário dos modelos que necessitam manter em memória uma janela com os últimos N frames para calcular a mediana, este modelo estima a mediana de forma recursiva e, portanto, somente precisa recordar a própria mediana anterior. Desta forma, torna-se especialmente interessante para aplicações com restrições de memória, caso típico de sistemas embarcados.

A variância dos pixels é estimada de maneira similar de acordo com a tabela 4.2, onde N é um parâmetro do algoritmo e Δ_t a diferença absoluta entre o valor do pixel no frame corrente e sua mediana estimada no tempo t. O valor de N é determinado empíricamente e na maior parte dos casos $N = 4$ mostrou-se adequado.

Para cada frame capturado
para cada pixel x
$\Delta_t(x) = |M_t(x) - I_t(x)|$

(1)

Inicialização
para cada pixel x
se $V_0(x) = \Delta_0(x)$

Para cada frame capturado
para cada pixel x
se $V_{t-1}(x) < N \cdot \Delta_t(x)$ então $V_t(x) = V_{t-1}(x) + 1$
se $V_{t-1}(x) > N \cdot \Delta_t(x)$ então $V_t(x) = V_{t-1}(x) - 1$

(2)

Tabela 4.2: Variância Σ-Δ

Finalmente, um pixel é marcado como *foreground* caso satisfaça a inequação mostrada na tabela 4.3, onde τ é um limiar dependente da aplicação. A utilização da variância nesta inequação tem a finalidade de aumentar ou diminuir a probabilidade de determinado pixel ser considerado *foregound*. Isso é feito para reduzir problemas com distratores (subseção 2.1.1), pois pixels pertencentes a estas regiões apresentam, tipicamente, maior variância e necessitarão de uma maior diferença Δ_t para serem marcados como foreground.

Para cada frame capturado
para cada pixel x
se $\Delta_t(x) > V_t(x) + \tau$
então $F_t(x) = 1$
senão $F_t(x) = 0$

Tabela 4.3: Decisão Σ-Δ

A última etapa da segmentação é a filtragem morfológica. Esta etapa consiste simplesmente em aplicar o operador de abertura (seção 2.1) no foreground extraído. Seu objetivo é eliminar pequenas regiões da imagem segmentadas como foreground que causam processamento desnecessário ou até mesmo confusão no rastreamento.

Algumas outras características listadas no capítulo anterior tornam este algoritmo interessante para implementação embarcada:

- os pixels são acessados sequencialmente,
- o algoritmo pode ser dividido em etapas, tal qual estágios de um pipeline,
- executa somente operações simples sobre grandes volumes de dados,
- somente executa instruções sobre valores inteiros.

Após o algoritmo implementado, observou-se que ele tem grandes dificuldades com o problema chamado *Sleeping Person* (subseção 2.1.1). A solução encontrada para isto foi uma realimentação do módulo de rastreamento para a segmentação. Esta realimentação consiste em uma máscara binária, indicando quais pixels não devem ser incorporados pelo modelo ao background.

4.2.2 Rotulação

O resultado da segmentação é uma imagem binária, na qual os pixels considerados *foreground* têm o valor '1' e todos os demais '0'. A rotulação dos componentes conexos possui dois objetivos principais: o primeiro é identificar e atribuir diferentes rótulos às diferentes regiões, correspondentes aos objetos, e o segundo é gerar uma representação de mais alto nível que facilite etapas posteriores do processamento.

O algoritmo utilizado neste trabalho é do tipo *two-pass* e agrupa regiões 8-conectadas. A imagem é percorrida pixel a pixel, da esquerda para direita e de cima para baixo, atribuindo rótulos iguais aos pixels adjacentes conectados e armazenando equivalências entre os rótulos atribuídos em um grafo, no qual os vértices correspondem aos rótulos e as arestas às equivalências. Então, as equivalências são resolvidas identificando-se os subgrafos conectados e uma segunda varredura atribui os rótulos finais. Ao longo desta segunda varredura, o vetor de características de cada objeto é extraído. Este vetor corresponde a uma estrutura de dados contendo o rótulo, o *bounding box* (menor retângulo que envolve o objeto), a área em pixels e a posição do centróide do objeto.

O resultado desta etapa é uma lista contendo os objetos presentes na cena e seus respectivos vetores de características.

4.2.3 Rastreamento

Uma vez que os objetos em movimento foram devidamente extraídos e caracterizados, eles devem ser rastreados ao longo dos frames. Esta etapa consiste em localizar os mesmos objetos presentes em diferentes frames, de forma que sua trajetória seja identificada. Para tal, são comparadas as características tanto espaciais quanto temporais dos objetos retornados pela rotulação com as dos objetos sendo rastreados em determinado momento. A implementação deste módulo foi baseada no algoritmo proposto por Li et al.[8].

Figura 4.4: Silhueta correspondente a duas pessoas

Cada região diferentemente rotulada é chamada de silhueta. Cada uma destas silhuetas, por sua vez, corresponde a um ou mais objetos presentes na cena (Figura 4.4). Isso acontece, principalmente, devido a imperfeições na segmentação. Para cada silhueta que corresponde a somente um objeto, o histograma de cores é calculado e adicionado ao seu vetor de características. Assumindo que os frames são capturados a uma taxa moderada ou alta (e.g. fps>8), o deslocamento dos objetos entre dois frames consecutivos deve ser pequeno, e por consequência, silhuetas presentes em frames consecutivos que pertençam ao mesmo objeto estarão sobrepostas. Partindo desta suposição, é construído um *grafo*

dirigido acíclico (DAG) que representa esta relação entre as silhuetas em frames consecutivos.

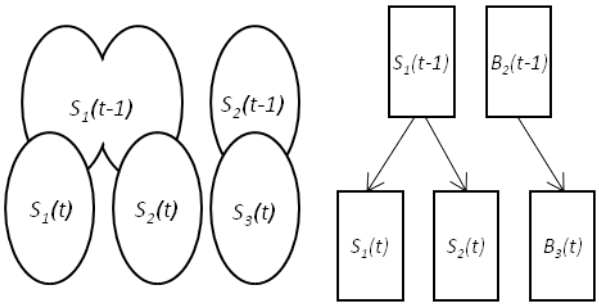

Figura 4.5: Representação da sobreposição de silhuetas em frames consecutivos através de um DAG

O DAG construído possui apenas dois níveis, chamados de pai e filho. Os vértices no nível pai representam as silhuetas na cena no tempo $t-1$, enquanto os vértices do nível filho correspondem as silhuetas presentes na cena no tempo t. Cada aresta deste grafo corresponde a uma sobreposição entre um vértice pai e um vértice filho, indicando as possíveis localizações dos objetos sendo rastreados no tempo t. Um exemplo de DAG é mostrado na figura 4.5.

O passo seguinte do algoritmo consiste em associar os objetos sendo rastreados às silhuetas presentes na cena no tempo t.

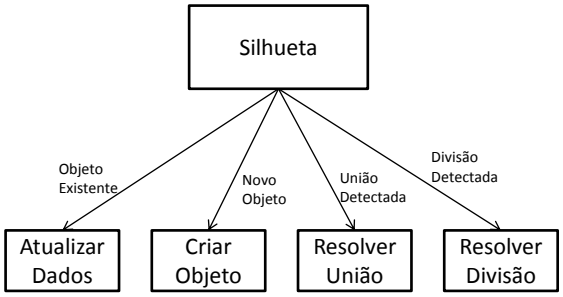

Figura 4.6: Problema da associação de objetos à silhuetas

No caso mais simples, um vértice pai está associado a somente um vértice filho. Neste caso não há nenhuma ambiguidade e todos os objetos associados ao vértice pai são associados ao vértice filho.

Se um vértice filho possuir dois ou mais pais, significa que temos objetos se unindo. Neste caso, todos os objetos atribuídos aos vértices pais são atribuídos aos vértices filhos. Quando um vértice pai possui dois ou mais filhos temos uma silhueta se dividindo (Figura 4.5). Neste caso, as características associadas aos objetos são utilizadas para resolver a ambiguidade. Compara-se o histogramas associados aos objetos com o histograma associado à silhueta através da distância de Bhattacharya e monta-se uma matriz com os resultados. Estes resultados são então comparados, e no caso de não haver uma correspondência satisfatória, ou manutenção da ambigüidade, são utilizados o vetor de velocidade e área para associação.

Caso um vértice filho não possua nenhum pai, há duas possibilidades: este vértice corresponde a um novo objeto ou a um problema na segmentação. Dois critérios são utilizados para resolver este problema. Uma vez que objetos não podem surgir em qualquer

lugar da imagem, um novo objeto é inicializado somente se a nova silhueta encontrar-se próxima de alguma borda da imagem. Além disso, este novo objeto é descartado caso não esteja presente no próximo frame do vídeo.

Após a etapa de atribuição de objetos às silhuetas, o algoritmo atualiza as informações referentes aos objetos sendo rastreados. A posição dos centróides dos objetos são guardadas para identificar a trajetória. Além disso, os histogramas de cores são atualizados e são estimados vetores de movimento para cada objeto baseando-se na diferença da posição de seus centróides entre dois frames consecutivos. Quando um objeto está formando um grupo, sua posição atual é estimada através de seu vetor de velocidade nos frames anteriores e o bounding box da silhueta a que corresponde.

4.2.4 Análise

A análise da posição dos objetos ao longo do tempo, bem como a interação entre eles pode ser utilizada para extrair informações acerca da cena. De fato, partindo-se da hipótese de que o software possui informação confiável a respeito da posição atual dos objetos e suas respectivas trajetórias, é relativamente simples de se derivar algumas informações.

O sistema implementado é capaz de contar pessoas que cruzam uma determinada linha virtual na cena (Figura 4.7). Para tal, ele compara a informação sobre em qual lado da linha o objeto se encontra no frame corrente e no frame anterior. Caso os lados sejam diferentes, o contador apropriado é incrementado. Para evitar que pessoas próximas a linha sejam contadas várias vezes, o contador somente é incrementado quando as mesmas cruzam totalmente a linha virtual.

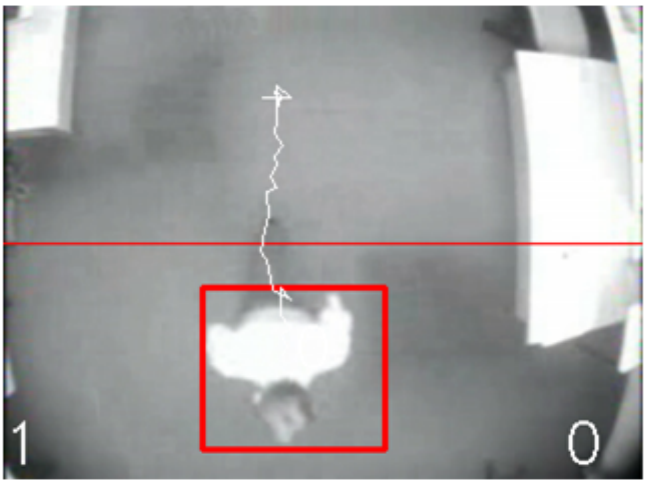

Figura 4.7: Contagem de pessoas em um corredor

Uma análise similar pode ser utilizada em corredores de desembarque de aeroportos. Normalmente, não se deseja que uma pessoa que já tenha desembarcado volte ao avião. Portanto, o sistema pode analisar o vídeo de maneira similar à contagem e emitir um alarme quando alguém cruzar a linha no sentido proibido.

Outro exemplo interessante é a detecção de objetos abandonados. Objetos abandonados são geralmente um risco potencial e uma preocupação constante, especialmente em aeroportos. Eles podem ser detectados, neste trabalho, identificando pequenas silhuetas que se separam de silhuetas maiores. A silhueta maior permanece parada, enquanto a maior afasta-se.

4.3 Resultados

A fim de avaliar a performance geral do sistema proposto, foram realizados alguns experimentos. Esses experimentos consistiram de sequências de vídeo em diferentes ambientes. Em todos eles, a câmera foi posicionada verticalmente a uma altura entre 3m e 6m. Requisitos como tempo de processamento são analisados, porém o objetivo principal desta etapa é verificar se a arquitetura e os algoritmos utilizados são capazes de atender aos requisitos funcionais pretendidos.

De modo geral o sistema mostrou-se adequado ao que se propõe, especialmente em ambientes internos e com boa iluminação. Os maiores problemas encontrados foram devido a imperfeições na segmentação. Por exemplo, sombras intensas causaram alterações na forma dos objetos, fazendo até mesmo com que dois objetos fossem segmentados como um.

Figura 4.8: Segmentação incorreta devido à sombra intensa

O principal desafio na etapa do rastreamento foi lidar com grupos e oclusões. A figura 4.9 exemplifica uma destas situações. A coluna à esquerda mostra a imagem capturada pela câmera e a coluna à direita o resultado da segmentação para três frames diferentes de um cenário de teste. O número acima do *bounding box* é um rótulo único atribuído a cada pessoa. Observa-se que em determinado momento as duas pessoas sendo rastreadas são segmentadas como uma única silhueta. Logo após, separam-se novamente e os rótulos são atribuídos corretamente. Em geral, situações como esta envolvendo duas ou três pessoas foram corretamente resolvidas. Porém, a partir de quatro pessoas o comportamento fica um pouco imprevisível.

A fim de obter-se uma avaliação do desempenho de cada módulo do sistema foi utilizado o *profiler* da GNU chamado *gprof*. Esse teste consistiu em gerar o perfil do processamento de uma sequência de vídeo contendo 711 frames com resolução de 320 x 240 pixels. Os resultados obtidos foram extremamente interessantes. Cerca de 94% do tempo de execução foi gasto na etapa de segmentação. Fica evidente que a segmentação é o grande gargalo deste sistema e que deve ser acelerado por um DSP ou FPGA em uma aplicação embarcada. O relatório completo gerado pelo *profiler* pode ser visto no anexo A.

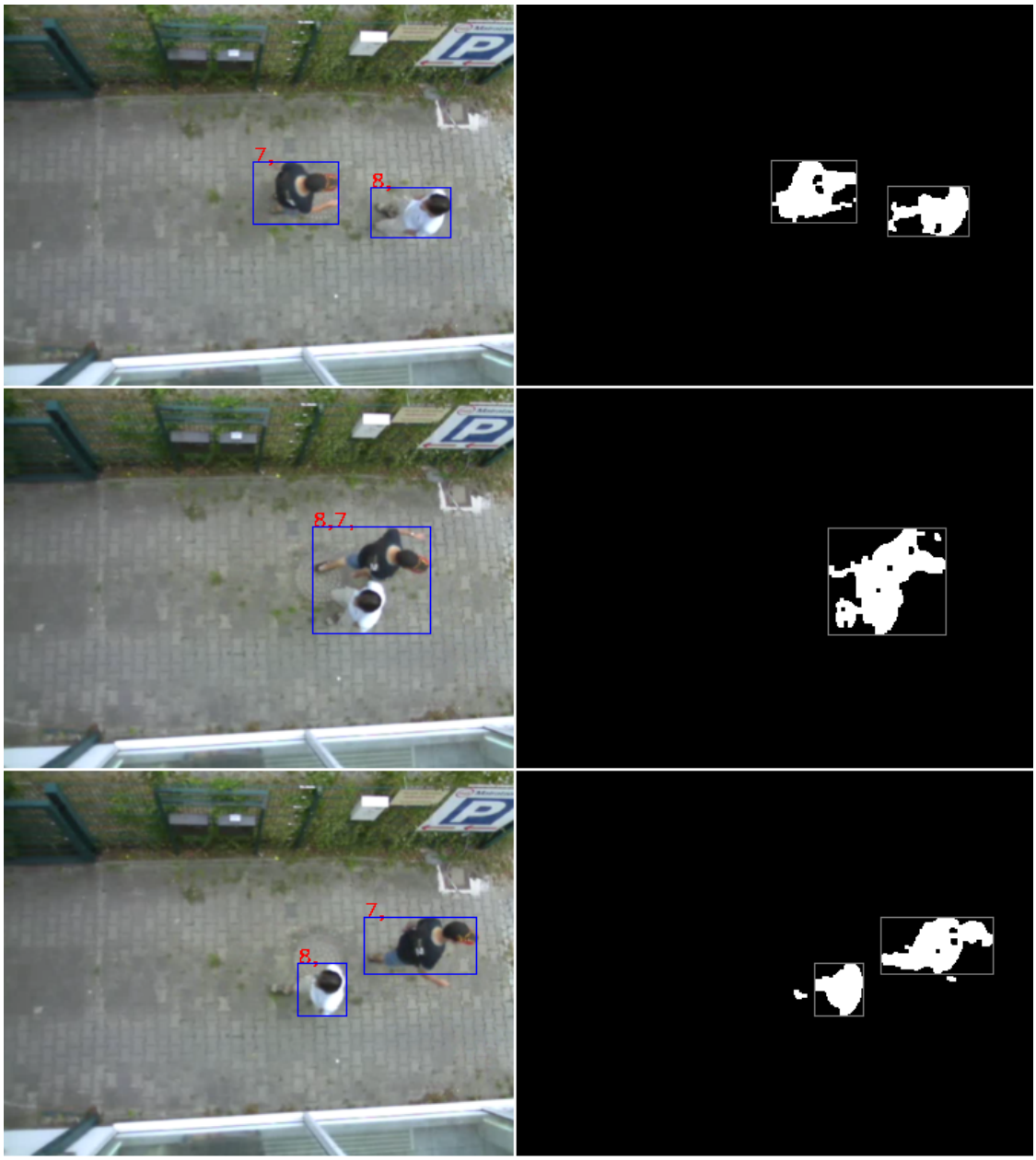

Figura 4.9: Rastreamento complexo

5 IMPLEMENTAÇÃO DAVINCI

Este capítulo apresenta a implementação embarcada do sistema descrito no capítulo anterior. Devido ao curto espaço de tempo disponível para o desenvolvimento, somente a etapa da segmentação foi realmente implementada nesta plataforma, porém a arquitetura descrita é adequada ao pipeline de processamento completo. A plataforma escolhida para isso é um processador pertencente à família DaVinci da *Texas Instruments*. Primeiramente o processador e o ambiente de desenvolvimento são apresentados. Em seguida, a arquitetura do software é detalhada e, finalmente, os resultados são avaliados.

5.1 Plataforma DaVinci

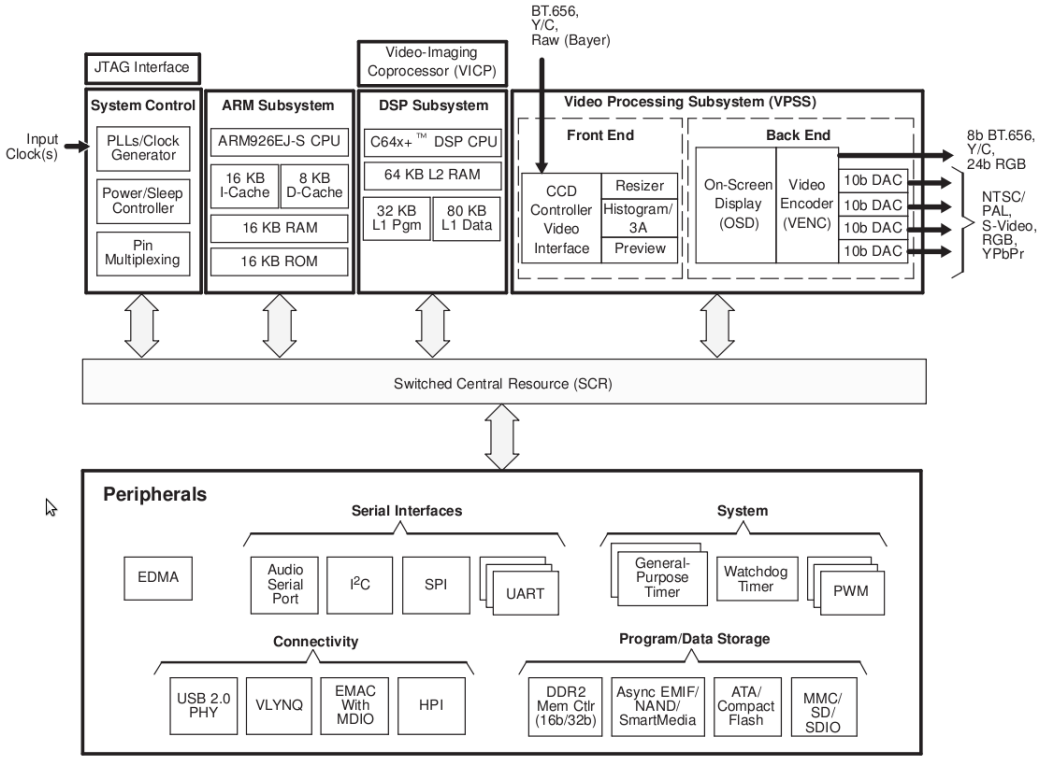

Figura 5.1: Diagrama do processador TMS320DM6446

DaVinci é uma família de processadores especialmente projetados para lidar com grandes fluxos de dados digitais em tempo real. Diferentemente de GPUs, são voltadas ao mercado de dispositivos embarcados, tais como *set-top boxes* e câmeras inteligentes. A

figura 5.1 mostra o diagrama do processador empregado neste trabalho. Ele é um dispositivo do tipo MPSoC, incorporando um DSP de alta performance C64x+, um processador RISC de propósito geral ARM926EJ-S, um subsistema de processamento de vídeo com uma série de hardwares específicos para funções como redimensionamento de imagens, cálculo de histogramas, exibição de texto na tela, conversão D/A, além de outros periféricos tais interfaces USB, serial e com memórias.

5.2 Ambiente de Desenvolvimento

O ambiente utilizado para implementação é o módulo de avaliação de vídeo digital DM6446 (DVEVM) da Texas. Este ambiente consiste em uma placa de desenvolvimento contendo o processador DaVinci TMS320DM6446 como elemento central, e o chamado DVSDK, que é um conjunto de componentes de software para facilitar e acelerar o desenvolvimento de aplicações para esta plataforma. O DVSDK inclui, por exemplo, os *cross-compilers* para o DSP e para o ARM, o Linux MontaVista Pro, u-boot e drivers para os periféricos existentes na plataforma.

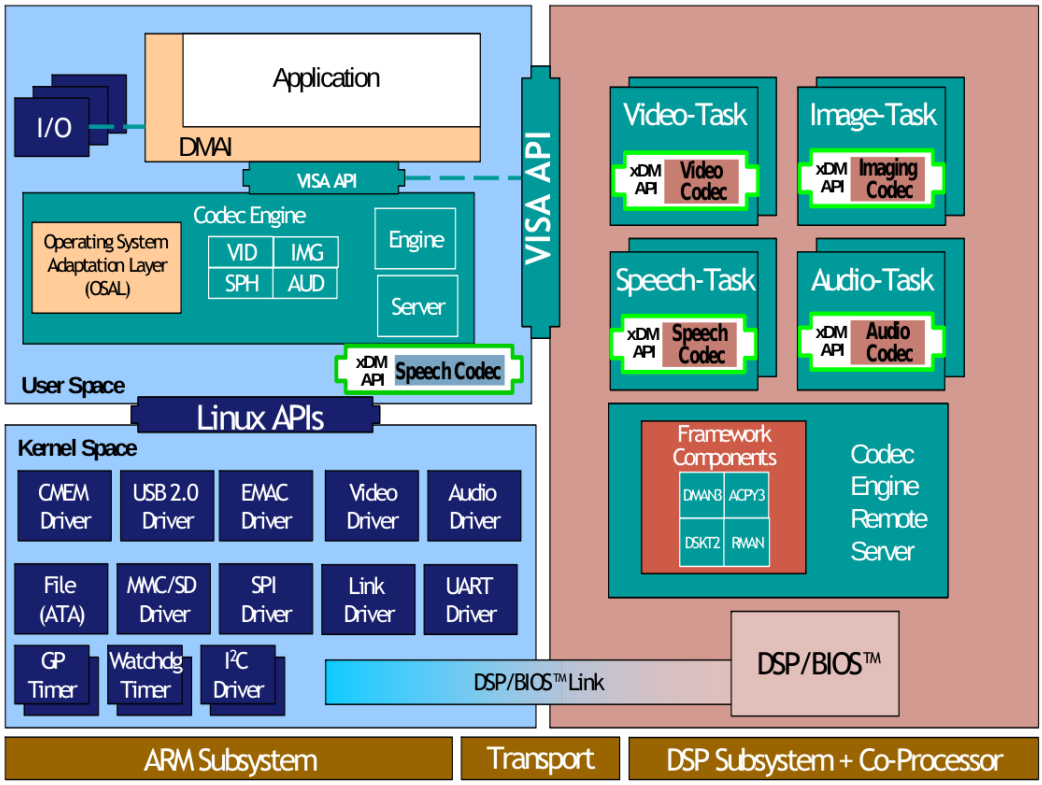

Figura 5.2: Componentes de software utilizados para desenvolvimento de aplicações com o DVEVM

A figura 5.2 mostra os componentes de software usados no desenvolvimento de aplicações no DVEVM. As aplicações são executadas no subsistema ARM, sobre um sistema operacional Linux, que disponibiliza um grande número de APIs, incluindo drivers e timers para a aplicação. O ARM é o responsável pelo controle geral do sistema, pelas operações de I/O e o processamento genérico das aplicações. Para processar sinais de audio, vídeo ou imagem são utilizadas as APIs VISA disponibilizadas pelo *Codec Engine*. Por sua vez, o *Codec Engine* utiliza os serviços fornecidos pelo *DSP/BIOS Link*

e protocolos como xDM para se comunicar com um servidor remoto sendo executado no subsistema DSP. O DSP processa o sinal e armazena os resultados em uma memória compartilhada que o ARM, então, pode acessar.

5.2.1 Algoritmos xDM

xDM é um padrão criado pela Texas Instruments para o desenvolvimento de algoritmos[1] para plataformas DSP. Sua principal motivação é facilitar a integração de algoritmos DSP, evitando custos de reengenharia e aumentando a reusabilidade de algoritmos desenvolvidos. Para um algoritmo estar de acordo com este padrão ele deve implementar a interface VISA especificada pelo padrão. Além disso, o algoritmo deve seguir uma série de regras. Por exemplo, algoritmos não podem acessar diretamente nenhum dispositivo periférico e o código deve ser totalmente realocável.

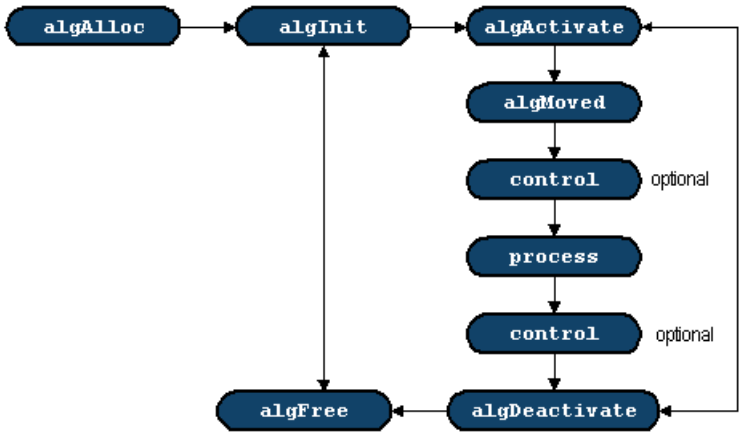

Figura 5.3: Sequência de execução válida das funções da interface xDM

A figura 5.3 ilustra uma sequência válida de execução de um algoritmo e as funções que são definidas na interface VISA e devem ser implementadas.

5.2.2 DSP/BIOS

DSP/BIOS é um kernel multitarefa de tempo real desenvolvido pela Texas Instruments especialmente para suas plataformas DSP. Ele consiste de um conjunto de serviços de tempo real na forma de bibliotecas de tempo de execução. Estes serviços fornecem a base para o desenvolvimento de aplicações. Os serviços incluem:

- um escalonador multitarefa preemptivo de tempo real.
- abstração do hardware.
- módulos de I/O para gerenciamento de streams de dados.
- funções que realizam em tempo real a captura de informações geradas pelo DSP durante a execução de programas.

[1] segundo a terminologia utilizada neste contexto pela Texas, algoritmo corresponde a um módulo que consome uma stream de dados, processando-a e resultando em outra stream.

5.2.3 DSP/BIOS Link

DSP/BIOS Link é o componente utilizado para comunicação entre os processadores, fornecendo uma interface genérica que abstrai as características da conexão física entre o ARM e o DSP. Ele não estabelece nenhuma restrição quanto ao sistema operacional rodando no ARM, porém como o nome sugere, ele espera que DSP/BIOS seja o sistema operacional do DSP. Os seguintes componentes fazem parte deste modulo:

PROC: fornece serviços para gerenciamento do DSP, tais como carregar um executável na memória, iniciar a execução a partir de determinado endereço e parar a execução.

POOL: fornece uma interface para configurar regiões de memória compartilhadas entre os processadores e sincronizar conteúdo de *buffers* vistos pelos dois processadores

NOTIFY: permite que aplicações notifiquem eventos para o processador remoto, além de receber notificações de eventos que ocorrem no processador remoto.

MPCS: permite a criação de seções críticas que forneçam acesso mutuamente exclusivo à estruturas de dados compartilhadas.

MPLIST: fornece um mecanismo de streaming de dados baseado em lista encadeada.

CHNL: fornece um canal lógico de transferência de dados no espaço da aplicação.

MSGQ: fornece um mecanismo baseado em fila para troca de mensagens curtas de tamanho variável.

RING IO: permite a criação de buffers circulares na memória compartilhada.

5.2.4 Codec Engine

O Codec Engine é a parte central do *framework* disponibilizado pela Texas. Ele pode ser considerado uma camada de abstração superior na comunicação entre os processadores, permitindo que chamadas de funções no DSP pareçam simples chamadas locais para o ARM. Esta abstração consiste em uma camada de software que implementa mecanismos de chamada remota de procedimentos (RPC). Para cada procedimento remoto, há uma função *stub* cliente e uma função *stub* servidora correspondente. Quando uma função que deva ser remotamente executada é invocada, a função *stub* cliente é executada. Esta, por sua vez, ao invés de executar a função esperada, empacota o comando e seus respectivos parâmetros em uma mensagem e a envia ao servidor. O servidor recebe esta mensagem e a encaminha à função *stub* servidora correspondente. Esta, desempacota o comando juntamente com seus parâmetros e realiza uma chamada local (da sua perspectiva) para a função requisitada. Quando o processamento da função acaba, a função *stub* servidora empacota o valor de retorno em uma mensagem e a retorna ao cliente. O *stub* cliente, por fim, recebe a mensagem, desempacota e repassa o valor de retorno à aplicação. A figura 5.4 ilustra este processo de chamada remota.

Uma vez que o Codec Engine implementa os *stubs* cliente e servidor para as funções definidas na interface VISA, qualquer algoritmo que siga o padrão xDM pode ser utilizado através de simples chamadas de função.

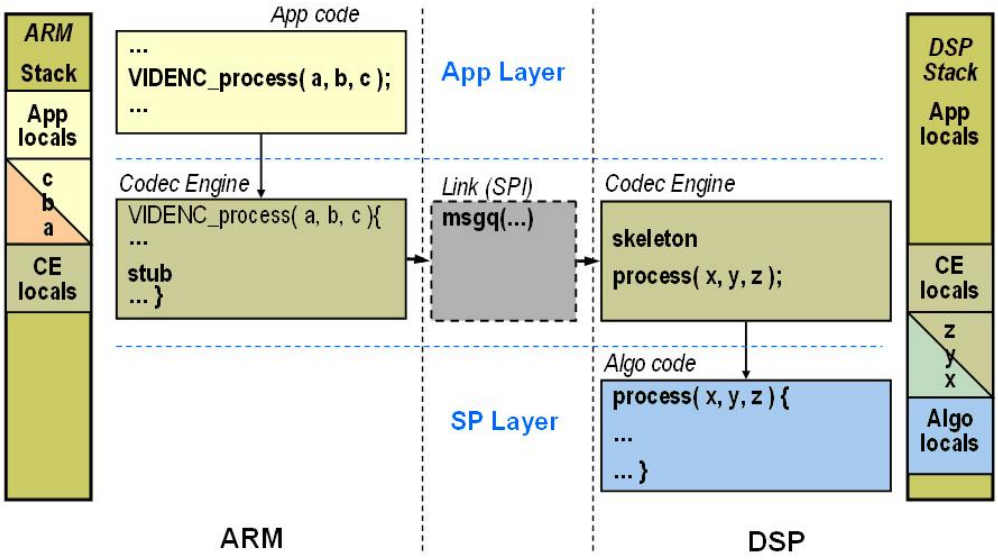

Figura 5.4: RPC no Codec Engine

5.2.5 CMEM

Para comunicação entre os processadores, o Codec Engine utiliza o módulo DSP/-BIOS Link e uma memória compartilhada. O mecanismo utilizado nesta comunicação é a troca de mensagens, permitindo que buffers grandes de dados sejam passados eficientemente através de ponteiros, uma vez que o buffer reside na memória compartilhada. Porém, esta arquitetura baseada em memória compartilhada apresenta um problema com relação ao uso de endereçamento virtual no ARM. O DSP presente na plataforma utilizada não possui MMU, e deste modo não é capaz de mapear segmentos de memória não-contiguos fisicamente em blocos logicamente contíguos.

O CMEM foi criado justamente para resolver este problema. Ele é executado no ARM como um módulo do kernel que permite a criação e gerenciamento de *pools* de memória. Blocos de memória fisicamente contíguos pertencentes a estes *pools* podem ser alocados em tempo de execução por uma aplicação através da API deste componente. Além disso, este módulo disponibiliza serviços de tradução de endereçamento virtual/físico. Desta forma, é importante que os buffers que necessitem ser acessados tanto pelo ARM quanto pelo DSP sejam alocados através deste módulo.

5.3 Arquitetura do Software

A arquitetura proposta para esta plataforma segue o mesmo pipeline detalhado no capítulo anterior, porém devido ao curto espaço de tempo disponível para o desenvolvimento, somente a etapa da segmentação foi implementada. A arquitetura descrita nesta seção, no entanto, é adequada ao pipeline de processamento completo.

A primeira diferença com relação à implementação para PC refere-se ao particionamento e mapeamento das tarefas para as unidades de processamento. Devido às caracterísiticas já discutidas anteriormente, como o acesso sequencial à memória e independência entre as operações, o algoritmo de segmentação é o candidato ideal para implementação no DSP. Deste modo, o ARM é liberado para executar as etapas de alto nível do pipeline de processamento, além das tarefas de controle do sistema.

A arquitetura proposta para o sistema é mostrada na figura 5.5. Do lado do ARM são

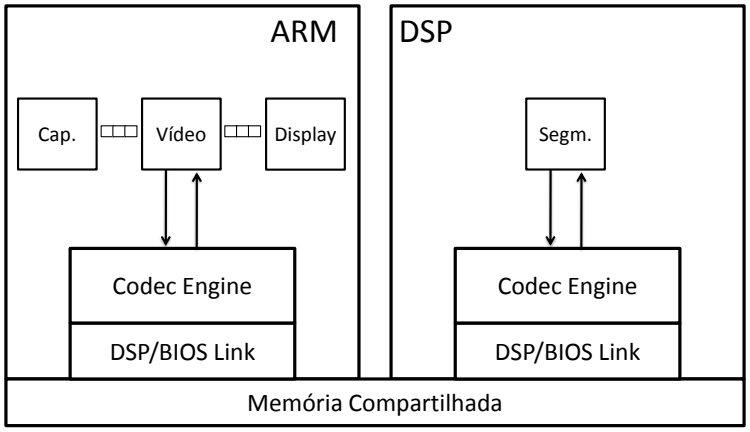

Figura 5.5: Arquitetura do Software

executadas três threads: a thread de captura, a thread de vídeo e a thread de display. Estas threads comunicam-se através de filas circulares de buffers. Do lado do DSP, apenas o algoritmo de segmentação é executado. A comunicação entre ambos os processadores se dá através de uma região compartilhada de memória. Toda a complexidade desta interação entre os processadores, no entanto, é abstraída pelo uso do Codec Engine, de modo que uma simples chamada de função no ARM é suficiente para executar a segmentação remotamente no DSP. A interação entre as threads durante o loop principal do sistema é mostrada na figura 5.6.

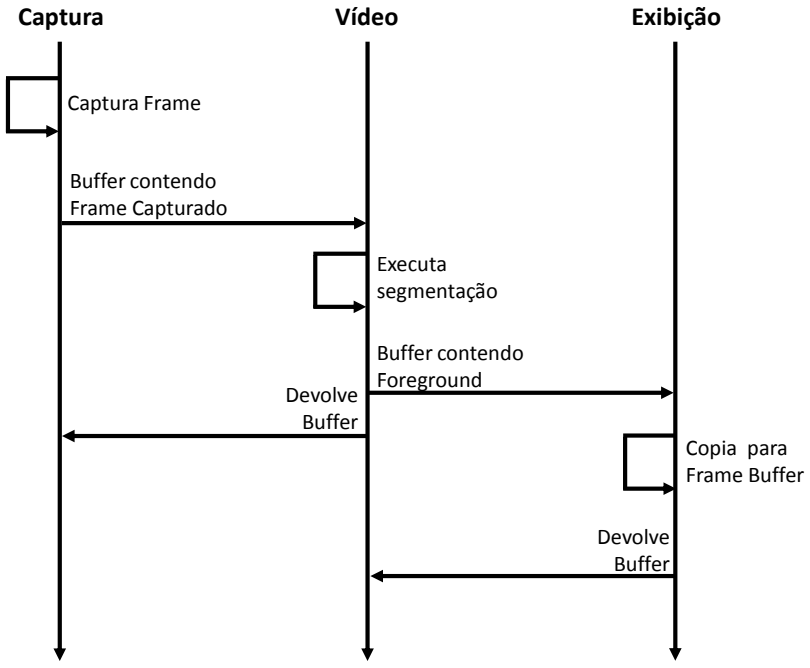

Figura 5.6: Interação entre as threads sem rastreamento

5.3.1 Thread de Vídeo

Esta é a principal thread do sistema. Sua execução pode ser dividida em duas etapas: inicialização do sistema e processamento do vídeo. A inicialização consiste em:

- Inicializar o Codec Engine e o DSP;
- Alocar, através do módulo CMEM, buffers contíguos de memória para troca entre as threads;
- Criar as threads de captura e exibição com os respectivos parâmetros.

Após a inicialização, esta thread executa um loop onde recebe da thread de captura um buffer contendo o frame capturado e da thread de display um buffer vazio, chama a segmentação remota passando como parâmetro os ponteiros para os buffers recebidos. Ao final do processamento, devolve o buffer contendo o frame capturado para a thread de captura e repassa o buffer com o resultado para a thread de exibição. É interessante ressaltar que as chamadas remotas feitas através do Codec Engine são bloqueantes e, portanto, durante o intervalo de tempo que a segmentação está sendo executada no DSP, o ARM está livre para a execução das outras threads.

5.3.2 Thread de Exibição

Esta thread é responsável pela exibição do resultado da segmentação. Sua tarefa consiste em copiar o buffer retornado pela thread de vídeo para o framebuffer do Linux. Para não consumir processamento do ARM durante esta cópia, é utilizado o módulo de redimensionamento de imagens (resizer) contido no subsistema de vídeo do processador. Esta cópia é implementada em outra thread para ser feita em paralelo com o processamento do vídeo.

5.3.3 Thread de Captura

A thread de captura é responsável pela inicialização do dispositivo de captura e posterior captura dos frames através de chamadas a este dispositivo. Esta captura consiste simplesmente em uma chamada ao driver de dispositivo. O frame retornado pelo driver de captura é copiado, utilizando o módulo de redimensionamento de imagens do subsistema de vídeo, para um buffer localizado na região de memória compartilhada e alocado utilizando o módulo CMEM.

5.3.4 Segmentação

O algoritmo utilizado para a segmentação é o mesmo detalhado no capítulo anterior. A implementação é feita de maneira similar, porém implementando as funções definidas na interface xDM e, desta forma, podendo ser utilizado em conjunto com o Codec Engine para chamadas remotas.

5.3.5 Pipeline Completo

No caso da implementação do pipeline completo, ela seria muito similar à implementação somente da segmentação. As diferenças são basicamente duas:

- a thread de vídeo dividida em duas: segmentação e rastreamento;
- realimentação do rastreamento para segmentação.

Dividindo-se a thread de vídeo em duas e adicionando-se outra fila de buffers entre estas duas threads, o rastreamento poderia ser executado no ARM, totalmente em paralelo com a segmentação no DSP.

A realimentação do rastreamento é a mesma comentada ao final da subseção 4.2.1. Ao invés de simplesmente devolver o buffer após utilizado, ele seria preenchido com uma máscara contendo as regiões da imagem que não deveriam ser adaptadas.

A interação entre as threads ficaria como mostrado abaixo:

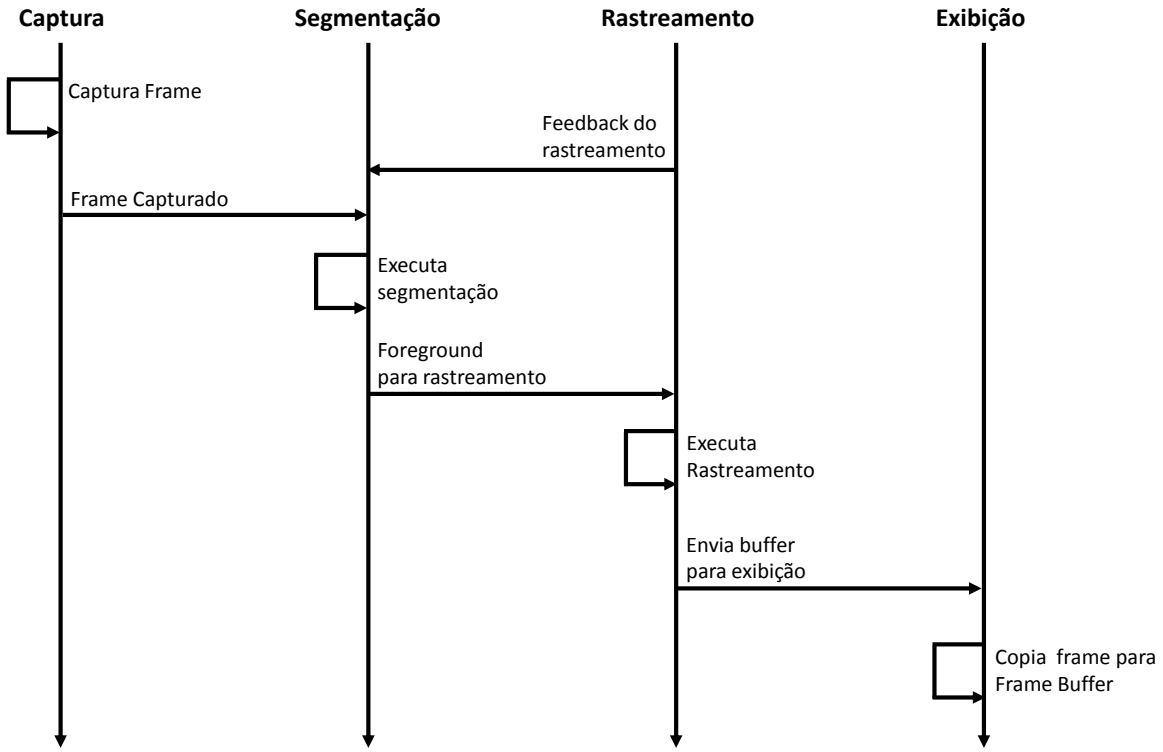

Figura 5.7: Interação entre as threads com rastreamento

5.4 Resultados

Foram executados testes, ligando-se diretamente uma câmera à plataforma de desenvolvimento e capturando-se frames com resolução de 720x480 pixels. Através de APIs fornecidas pelo Codec Engine, observou-se que a execução do programa consome em média 70% do núcleo DSP e 3% do ARM, processando 20 frames por segundo. Este fato é importante, pois mostra que o ARM está praticamente livre para a implementação das etapas de rastreamento e análise do pipeline.

Os resultados obtidos na segmentações resultantes do DaVinci e do PC parecem similares, porém carecem de uma comparação mais precisa, principalmente para se avaliar o impacto da taxa de frames maior obtida no DaVinci. É importante afirmar também que o algoritmo implementado no DSP não está completamente otimizado. Desta forma, é possível implementar-se algoritmos mais sofisticados de segmentação, ou até mesmo outras etapas do processamento serem incorporadas ao DSP mediante otimizações em seu código.

Outro fato que deve ser ressaltado é que além do *framework* de desenvolvimento disponibilizado pela Texas Instruments, a documentação dos componentes e quantidade de exemplos encontrados é extensa. Todos estes fatores facilitam e aceleram o desenvolvimento de aplicações voltadas à esta plataforma.

6 CONCLUSÃO

Este trabalho apresenta um estudo de técnicas e o desenvolvimento de uma aplicação voltada à vigilância inteligente. Apesar das simplificações que se fizeram necessárias, os objetivos foram atingidos com sucesso. A sequência de processamento proposta é capaz de identificar objetos em movimento no campo de visão da câmera e segui-los através das cenas.

Em função de ser um projeto extenso, diversas melhorias foram pensadas, entre elas estão:

- detecção de sombras;
- inclusão de modelo de aparência de objetos;

Sombras foram o maior problema enfrentado pela segmentação. Elas causaram alterações na área e forma dos objetos, modificando suas características e, consequêntemente, confundindo a etapa de rastreamento. Alguns métodos para segmentar explicitamente a sombra foram vistos, entretanto, deixaram de ser implementados devido a outras prioridades do projeto.

A etapa de rastreamento não utiliza nenhum tipo de modelo de aparência para os objetos segmentados, deste modo, qualquer grupo de pixels descrevendo um movimento coerente será rastreado como um objeto de interesse. Incluir algum modelo para identificação de pessoas é importante para algumas aplicações deste sistema, como contagem de pessoas.

Desenvolvimento de software para plataformas SoC como o DaVinci é extremamente complexo. A obtenção de implementações eficientes de algoritmos, principalmente para o núcleo DSP, envolve pesquisa e conhecimento que poderiam também vir a ser tema de um outro trabalho.

Alguns desafios interessantes para o futuro, além da implementação do pipeline completo do processamento na plataforma DaVinci, seriam a detecção automatica de eventos mais complexos através de aprendizagem de máquina e desenvolvimento de sistemas de vigilância automatizados empregando múltiplas câmeras de forma cooperativa.

Finalmente, o projeto foi extremamente desafiante para o autor, permitindo o aprofundamento em diversas áreas abordadas durante o curso de Engenharia de Computação.

REFERÊNCIAS

[1] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, Cambridge, MA, 2008.

[2] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati. Detecting moving objects, ghosts, and shadows in video streams. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(10):1337–1342, oct. 2003.

[3] Luis M. Fuentes and Sergio A. Velastin. People tracking in surveillance applications. *Image Vision Comput.*, 24(11):1165–1171, 2006.

[4] I. Haritaoglu, D. Harwood, and L.S. Davis. W4: Who? when? where? what? a real time system for detecting and tracking people. In *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*, pages 222–227, 14-16 1998.

[5] D. Koller, J. Weber, T. Huang, J. Malik, G. Ogasawara, B. Rao, and S. Russell. Towards robust automatic traffic scene analysis in real-time. In *Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision Image Processing., Proceedings of the 12th IAPR International Conference on*, volume 1, pages 126–131 vol.1, 9-13 1994.

[6] M. Kölsch and S. Butner. *Hardware Considerations for Embedded Vision Systems*, pages 3–26. SpringerLink, 2009.

[7] Longin Jan Latecki and Roland Miezianko. Object tracking with dynamic template update and occlusion detection. *Pattern Recognition, International Conference on*, 1:556–560, 2006.

[8] Liyuan Li, Weimin Huang, I.Y.-H. Gu, Ruijiang Luo, and Qi Tian. An efficient sequential approach to tracking multiple objects through crowds for real-time intelligent cctv systems. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 38(5):1254 –1269, oct. 2008.

[9] Antoine Manzanera and Julien C. Richefeu. A new motion detection algorithm based on sigma-delta background estimation. *Pattern Recognition Letters*, 28(3):320–328, 2007.

[10] O. Masoud and N.P. Papanikolopoulos. A novel method for tracking and counting pedestrians in real-time using a single camera. *Vehicular Technology, IEEE Transactions on*, 50(5):1267–1278, sep 2001.

[11] C Phillips. A review of cctv evaluations: Crime reduction effects and attitudes towards its use. In *Surveillance of Public Space: CCTV, Street Lighting and Crime Prevention. Crime Prevention Studies*. Criminal Justice Press, 1999.

[12] M. Rossi and A. Bozzoli. Tracking and counting moving people. In *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, volume 3, pages 212–216 vol.3, 13-16 1994.

[13] Sankalita Saha. *Design methodology for embedded computer vision systems*. PhD thesis, College Park, MD, USA, 2007. Adviser-Bhattacharyya, Shuvra S.

[14] C. Stauffer and W.E.L. Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2, page 252 Vol. 2, 1999.

[15] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers. Wallflower: principles and practice of background maintenance. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 1, pages 255–261 vol.1, 1999.

[16] E Wallace and C Diffley. Cctv: Making it work," police scientific development branch of the home office (psdb) publication 14/98, 1998.

[17] Brandon C. Welsh and David P. Farrington. Effects of Closed-Circuit Television on Crime. *The ANNALS of the American Academy of Political and Social Science*, 587(1):110–135, 2003.

[18] C.R. Wren, A. Azarbayejani, T. Darrell, and A.P. Pentland. Pfinder: real-time tracking of the human body. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(7):780–785, jul 1997.

[19] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Anal. Appl.*, 12(2):117–135, 2009.

[20] Changjiang Yang, Ramani Duraiswami, and Larry Davis. Fast multiple object tracking via a hierarchical particle filter. In *ICCV '05: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, pages 212–219, Washington, DC, USA, 2005. IEEE Computer Society.

ANEXO A PROFILING DA IMPLEMENTAÇÃO PARA PC

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 94.05      9.01     9.01      711    12.67    12.67  updateBFModel(BFModel*, _IplImage*)
  4.18      9.41     0.40      711     0.56     0.62  objectLabel(_IplImage*,  trackable_t*)
  0.52      9.46     0.05      711     0.07     0.18  processTracking(_IplImage*, tracking_t*)
  0.52      9.51     0.05      668     0.07     0.07  numObjects(blob_s*)
  0.21      9.53     0.02      782     0.03     0.03  overlaped(rect_t, rect_t)
  0.16      9.54     0.01  1237320     0.00     0.00  addEq(eqMatrix_s*, int, int)
  0.10      9.55     0.01     5462     0.00     0.00  initEqLabel(eqMatrix_s*, int)
  0.10      9.56     0.01      711     0.01     0.04  generateDAG(trackable_t*, tracking_t*)
  0.10      9.57     0.01      711     0.01     0.01  calcEqLabels(eqMatrix_s*, foregroundAreas_t*)
  0.05      9.58     0.01      711     0.01     0.01  destroyList(eqMatrix_s*)
  0.00      9.58     0.00    17846     0.00     0.00  cvPoint(int, int)
  0.00      9.58     0.00     5462     0.00     0.00  destroyNode(neighbor_s*)
  0.00      9.58     0.00     5462     0.00     0.00  addNeighbors(eqMatrix_s*, neighbor_s*, int*)
  0.00      9.58     0.00     1165     0.00     0.00  cvScalarAll(double)
  0.00      9.58     0.00      994     0.00     0.00  cvRect(int, int, int, int)
  0.00      9.58     0.00      932     0.00     0.00  newOverlap(float, blob_s*)
  0.00      9.58     0.00      722     0.00     0.00  captureFrame(CvCapture*)
  0.00      9.58     0.00      454     0.00     0.00  deleteBlob(blob_s*)
  0.00      9.58     0.00      454     0.00     0.00  updateData(_IplImage*, blob_s*)
  0.00      9.58     0.00      454     0.00     0.00  bestFit(List*)
  0.00      9.58     0.00      454     0.00     0.00  newBlob()
  0.00      9.58     0.00      366     0.00     0.00  newAssignment(int, blob_s*, blob_s*)
  0.00      9.58     0.00       86     0.00     0.00  updateHistogram(_IplImage*, blob_s*)
  0.00      9.58     0.00       86     0.00     0.00  cvCalcHist
  0.00      9.58     0.00       19     0.00     0.00  deleteObject(object_s*)
  0.00      9.58     0.00       19     0.00     0.00  newObject(int)
  0.00      9.58     0.00       14     0.00     0.00  emptyObjectsList(blob_s*)
  0.00      9.58     0.00        1     0.00     0.00  initBFModel(BFModel*, _IplImage*)
  0.00      9.58     0.00        1     0.00     0.00  initTracking(_IplImage*, tracking_t*, int)
  0.00      9.58     0.00        1     0.00     0.00  cvSize(int, int)

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.
```

```
name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.
```

ANEXO B TRABALHO DE GRADUAÇÃO I

A smart-camera based system for surveillance applications

Alfredo Gaubert Capella Junior
Institute of Computer Science
University of Potsdam
gaubertc@cs.uni-potsdam.de

Cristophe Bobda
Institute of Computer Science
University of Potsdam
bobda@cs.uni-potsdam.de

Abstract

In this paper we propose a hardware/software co-design of a real-time tracking system for surveillance applications. The system is able to detect people and track their movement around a monitored area using a single camera vertically mounted in order to avoid fully overlapped people. Moreover, experimental results are presented.

1 Introduction

In security and also in marketing industry it is very interesting to know people flow information as well as which ways people are taking from a point to another. Besides that, at many public places, like an airport, we need to make sure people are kept away from some areas. Hence, the use of CCTV surveillance systems has grown enormously last years[1]. As a consequence of such a growth, more and more visual information is generated for these surveillance systems, becoming impossible for most organizations to use human resource to monitor all this information. Furthermore, human is error-prone due to fatigue or negligence. Therefore, it is important to develop accurate automatic surveillance systems.

Much effort has been done recently in such systems [1][2]. However, most of these focus on a PC-based approach [2][4][5]. Along with that, DSPs and FPGAs are getting faster and faster, allowing intelligent cameras based on it to perform complex tasks where only traditional PC-based approaches were suitable before. If all the processing can be done inside the camera, the result might be only a few hundred bytes of data to be sent somewhere. Using this kind of approach, we can shift from a central paradigm to a distributed control surveillance system. In this paper, we propose a real-time tracking system suitable for a smart camera.

This paper is organized as follows. Our smart camera is presented in section 2. In section 3, we describe our approach. First, partitioning the system in modules and detailing them afterwards. Finally, we present some experimental results and concluding remarks in section 4.

2 Hardware

Figure 1

We implemented the algorithm in our smart camera. Although the complete description of the camera architecture goes beyond the scope of this paper we will point out the most important parts of the hardware.

The core of the camera is a Virtex-4 FX FPGA from Xilinx. We implemented a streaming data interface (SDI) which is in detail explained in [6]. The core component is the SDI controller which reads data from a SDI channel or can write data to a SDI channel. It is directly connected to the memory of the system. The purpose is to handle the image data without interference from the CPU. A SDI channel consists of cascaded processing units (PU). They either initiate a new stream, like cameras, modify an existing stream, like our segmentation PU, or they read from a stream like a VGA module.

On top of the hardware we put an operating system, Linux in this case. It's running Intel's OpenCV library to easily port code developed on a standard PC to the smart cam-

era. To interact with the data processed by the PUs we can use the SDI controller. It is the central point where all image data is passing by. We only need to write a driver to access it from software side.

We implemented a segmentation PU to immediately separate the image in static background and moving foreground objects while it is getting streamed from the camera to the memory. The details of the segmentation are explained in section 3.1. To implement it in the FPGA we could use the DSP slices of the Virtex-4. They are able to do operations like multiplications in one clock cycle only. To update the background image we added a second SDI controller which creates a second stream containing the background. The segmentation PU adapts the background according to the current image, sends it back to the second SDI controller and that rewrites it to the memory.

To be able to read the foreground image in OpenCV we use the driver of the SDI controller. With that we know the position of the image data in memory and can easily use it directly in OpenCV. We only need to let the *IPLImage's data* point to that position. The remaining implementation keeps the same.

3 Proposed system architecture

Figure 2 shows the flow chart of the proposed system. The first steps of the system are background maintenance and foreground estimation. These steps result in a binary image containing foreground objects. Then, shadow pixels are removed from it. Next, we extract the blobs and its features from the binary image, generating a higher level representation of the blobs to the tracking module, which generates the trajectory of the object over time by locating its position in every frame of the video. Finally, the interaction between blobs and their trajectory is analyzed in order to trigger an alarm when objects are abandoned, or obtain statistical data, as number of people taking some direction. As depicted in Figure 2, we implemented the two lowest level steps of the system in hardware to exploit the inherent parallelism of such image processing algorithms.

3.1 Segmentation

The extraction of moving objects is the first step for most computer vision applications. The overall performance of the whole system depends on how accurately moving objects are extracted. It is usually done by three basic approaches: Optical flow, background subtraction and frame differencing[7]. Despite being the most robust of these techniques, optical flow requires large amount of processing, what is not available in the context of this work. Background subtraction can extract all moving pixels if a perfect background model is available. However, illumination variation, shadows, movement of background objects or even slight oscillations of the camera can interfere with the result accuracy [9]. Frame differencing does not make any assumption about the background and is used for dynamic environments. It suffers from the problem of foreground aperture being necessary infer movement from borders movement.

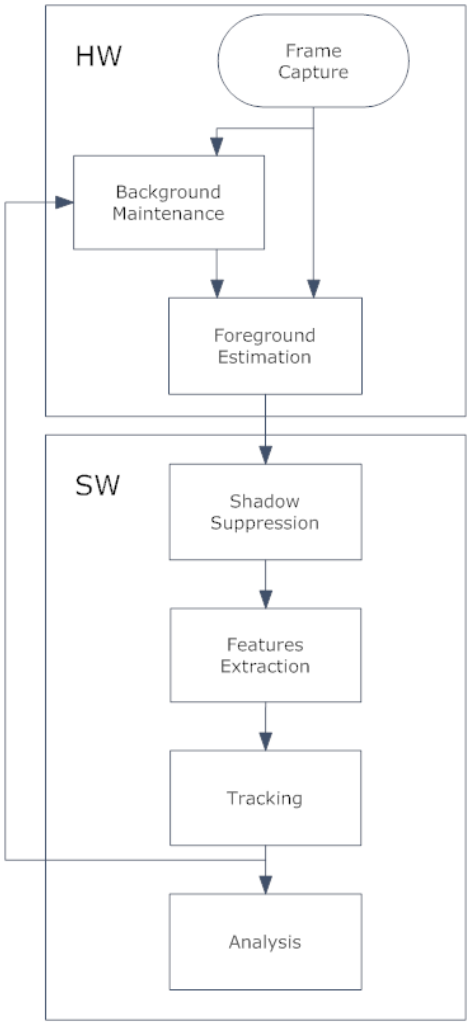

Figure 2

Our proposed system extracts moving objects using an algorithm based on Σ-Δ background estimation proposed by Manzanera and Richefeu [8]. This algorithm works as a digital conversion of a time-varying analog signal using Σ-Δ modulation, estimating the background using just comparisons and elementary increment or decrement operations, hence extremely cheap computationally. Besides that, it just keeps in memory the background estimated and a variance for each pixel, therefore lightweight in terms of memory too. Unlike [8], we use information from the three RGB channels to compute the background as we need this color information for the shadow suppression module. In order to avoid foreground objects that remain stopped with-

in the scene for many frames being added to background model, the segmentation receives a feedback from tracking module. This feedback is a mask image where pixels that should not be adapted are set to '1'.

for each channel in {R,G,B}
  for each pixel
    $B_t = B_{t-1} + sgn(I_t - B_{t-1})$
    $\Delta_t = B_t - I_t$
    if $\Delta_t \neq 0$
      $V_t = V_{t-1} + sgn(\Delta_t - V_{t-1})$
    if $\Delta_t < V_t + \tau$
      $F_t = 0$
    else
      $F_t = 1$

Table 1

Table 1 shows the general behavior of the segmentation algorithm. The $sgn(x)$ function is defined as $sgn(x) = 1$ if $x > 0$, $sgn(x) = -1$ if $x < 0$ and $sgn(x) = 0$ if $x = 0$, B_t represents current estimated background, Δ_t the difference between current frame and the estimated background, V_t the variance of pixel value, F_t the estimated foreground and τ an empirically defined threshold.

At the end, the foreground mask is processed by a morphological opening operator to remove small clusters in the image.

3.2 Shadow Suppression

Shadows represent a big challenge for accurate moving objects tracking. The difficulties associated with shadow detection arise since they differ significantly from the background and have the same motion as the objects casting them. They can cause object merging, object shape distortion, and even object losses (due to the shadow cast over another object). Many works have been published on shadow detection and suppression. In [9] a comparative evaluation of some existing approaches is presented.

In order to solve this problem, we used an algorithm like the proposed by Cucchiara et al. [10]. The algorithm uses Hue-Saturation-Value (HSV) color space that corresponds closely to the human perception of color and reveals more accuracy in distinguishing shadows than the RGB color space. It is based on the observations that a shadow makes the region it covers darker, but does not change its hue significantly and a shadow often lower the saturation of the background points. Consequently, the shadow points are classified by the following decision rule:

$$S(x,y) = \begin{cases} 1, & \alpha \leq \dfrac{B^V(x,y)}{I^V(x,y)} \leq \beta \\ & \wedge\, (I^S(x,y) - B^S(x,y) \leq \tau_S) \\ & \wedge\, (I^H(x,y) - B^H(x,y) \leq \tau_H) \\ 0, & otherwise \end{cases}$$

where I(x,y) and B(x,y) are the pixel values at coordinate (x,y) in the input image and in the background model, respectively. We apply this rule for each pixel considered foreground by the temporal segmentation.

3.3 Blobs and features extraction

After the segmentation process we have a binary image of the foreground regions and a connected component labeling is performed in this image to extract the blobs. During this extraction some useful features like area, bounding box, centroid and density (blob area divided by bounding box area) are computed for each blob. A list of blobs in current frame is passed to the tracking module.

3.4 Tracking

Once the moving objects have been extracted, they must be tracked across video frames. Our proposed system uses a combination of spatial and temporal features to perform the blobs matching between frames. The most challenging task in this step is to track correctly objects even after blobs joining together and splitting again.

Each foreground region in the segmented image is a blob. Each blob can be either one person or a group of people. For each blob that represents just one person, we include its color histogram to the extracted features.

Assuming that the frames are captured with a moderate or high frame rate (e.g. fps>8), the interframe displacements of a blob are small and consequently its corresponding regions in consecutive frames will overlap each other. From this assumption we build a directed acyclic graph (DAG) to represent this relation between blobs in consecutive frames[5].

The DAG contains two layers – the parent layer and the child layer. The nodes in the parent layer represent regions in the previous frame and the nodes in child layer represent the blobs in the current frame. Each edge in this graph represents an overlap between a blob in parent layer and a blob in child layer. This DAG provides us temporal information for matching blobs. Figure 3 shows an example of blobs in two consecutive frames and the resulting DAG.

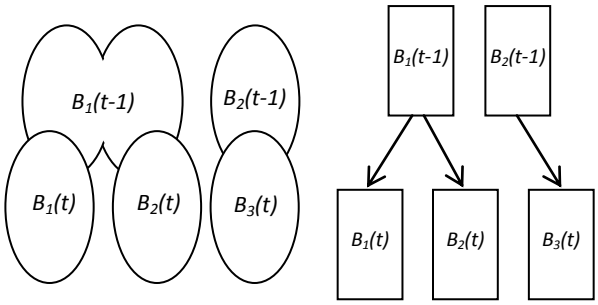

Figure 3

The next step in the tracking algorithm is the assignment. In this step, all objects from parent blobs should be assigned to child blobs. The simplest case is when a parent blob overlaps only one child blob. It means that all objects from parent blob should be assigned to the child blob. If a parent blob has two or more children blobs, we have a blob splitting and the extracted features are used to determine the correct assignment. Firstly, the algorithm tries to assign objects to child blobs containing just one object, as we can use a very reliable matching: histogram comparison. Bhattacharyya distance has proved to be efficient for such applications [5], being therefore used to compare the color histograms. Remaining objects are assigned to child blobs using speed direction and area variation between frames. Information about this split is stored for future use. Then, we deal with new and disappearing objects. If a child blob does not overlap any blob from previous frame, it is likely to be a new object. Since new objects can not appear anywhere, we must ensure that this blob is close to some image border, initialize a new object with a new ID number and assign it to the blob. However, if a parent node has no child, its objects are considered disappeared.

After the assignment step, the algorithm updates blobs and objects information. We keep the position of the blob centroid to form a trajectory when the blob is being tracked. Whenever multiple objects merge into a blob, we interpolate their position using the speed from previous frames and the bounding box of the blob.

3.5 Event detection

The analysis of blobs position and interaction can be used to detect some events.

Our proposed system is able to count people that cross a virtual line. In fact, as long as we have reliable information about blobs position, the counting implementation is very simple. We just need to keep information about in which side of the counting line the blob was in previous frame and which it is current frame, if sides are different we increment the number of people assigned in that blob to the appropriated counter. We just increment appropriated counter if the whole blob crosses the line, so that we avoid people close to counting line being counted several times.

Besides that, we can identify probable abandoned objects analyzing blobs splitting data. We can trigger an alarm when we observe one blob splitting into two blobs, one of these presenting no movement, and another moving away from the first.

Furthermore, additional information as intrusion in forbidden areas can be easily obtained from current implementation.

4 Experiments and conclusion

Experiments were performed to demonstrate the overall performance of the proposed system. These experiments were carried out using several real test sequences with the camera set 6m above the floor. The implemented system obtained interesting results operating at 10 frames per second, being able to track people successfully even if they merge into a group and split again, showing its suitability for surveillance applications. Figure 4 exemplifies such situation. The left column in this figure shows the image captured by the camera and the right column the result of segmentation for three different frames from a test scenario. We can observe in Figure 4 that the same labels are assigned to each person before and after they merge.

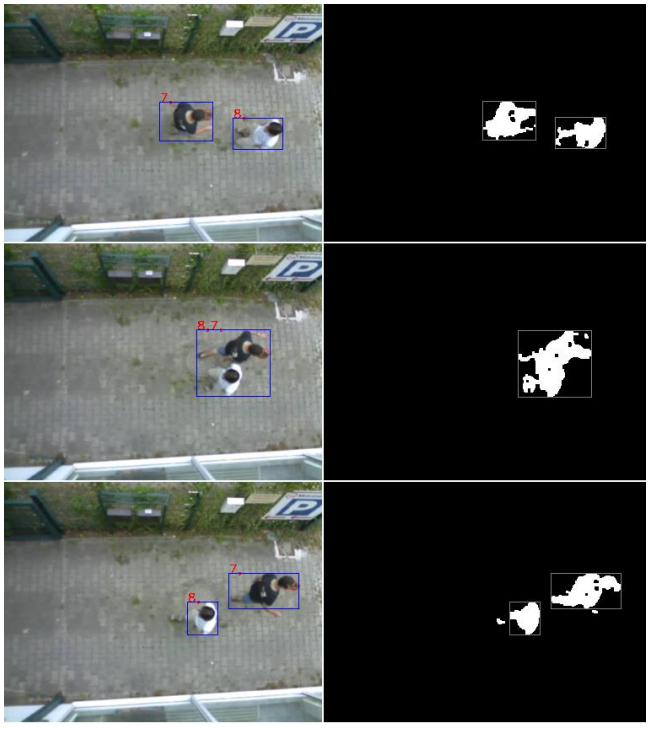

Figure 4

References

[1] Tracking People for Automatic Surveillance Applications

[2] Real-time Vision-based People Counting System for the Security Door

[3] “Fast Multiple Object Tracking via a Hierarchical Particle Filter,”.

[4] “W4: Real-Time Surveillance of People and Their Activities,”

[5] L.Li, W. Huang, I.Y.H.Gu, R. Luo, Q.Tian, "An efficient sequential approach to tracking multiple objects through crowds for real-time intelligent CCTV systems", *IEEE trans. Systems, Man and Cybernetics*, part B, vol.38, no.5, pp.1254-1269, 2008.

[6] F. Mühlbauer, L.O. Marchioro and C. Bobda. “Hardware Accelerated OpenCV on System on Chip”, *Reconfigurable Comunication-centric Systems-on-Chip Workshop*, 2008.

[7] B. Boufama, M.A. Ali, “Tracking multiple people in the context of video surveillance,” *in ICIAR*, 2007, pp. 581-592.

[8] A. Manzanera, J. Richefeu, “A robust and computationally efficient motion detect algorithm based on Σ-Δ background estimation”. *Pattern Recognition Letters*, vol. 28, no. 3, pp. 320-328, Feb. 2007.

[9] A. Prati, I. Mikic, M. M. Trivedi, et al, “Detecting moving shadows:algorithms and evaluation”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 7, pp. 9189-923, Jul. 2003.

[10] R. Cucchiara, C. Grana, M. Piccardi, A. Prati, et al, “Improving Shadow Suppression in Moving Object Detection with HSV Color Information,” *Proc. IEEE Int’l Conf. Intelligent Transportation Systems*, pp. 334-339, Aug. 2001.

[11] T. Horprasert, D. Harwood, and L.S. Davis, “A Statistical Approach for Real-Time Robust Background Subtraction and Shadow Detection,” *Proc. IEEE Int’l Conf. Computer Vision*, *FRAME-RATE Workshop*, 1999.

[12] http://www.youtube.com/watch?v=9X3j1Q8EACk