UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SISTEMA DE TRACKING DE OBJETOS A PARTIR DE VÁRIAS CÂMERAS

ANDRÉ LUÍS BELING DA ROSA

BLUMENAU
2010

2010/2-05

ANDRÉ LUÍS BELING DA ROSA

SISTEMA DE TRACKING DE OBJETOS A PARTIR DE VÁRIAS CÂMERAS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Dalton Solano dos Reis, M. Sc. - Orientador

BLUMENAU
2010

2010/2-05

SISTEMA DE TRACKING DE OBJETOS A PARTIR DE VÁRIAS CÂMERAS

Por

ANDRÉ LUÍS BELING DA ROSA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Dalton Solano dos Reis, M. Sc. – Orientador, FURB
Membro:	Prof. Paulo César Rodacki Gomes, Dr. – FURB
Membro:	Prof. Miguel Alexandre Wisintainer, M. Sc. – FURB

Blumenau, 07 de dezembro de 2010

Dedico este trabalho a minha família, que sempre esteve comigo.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que me apoiaram e incentivaram tornando possível o desenvolvimento deste trabalho.

Ao professor Dalton Solano dos Reis pela sua orientação e por ter acreditado na conclusão deste trabalho.

Aos colegas Maicon Rafael Zatelli, Samuel Yuri Deschamps e Victor Arndt Mueller, que foram companheiros durante todo o curso e com certeza contribuíram para a aquisição dos conhecimentos necessários para o desenvolvimento deste trabalho.

Se eu vi mais longe, foi por estar de pé sobre ombros de gigantes.

Sir Isaac Newton

RESUMO

Este trabalho apresenta um sistema de *tracking* capaz de fazer acompanhamento de objetos através de várias câmeras. A utilização de câmeras de vigilância é uma prática cada vez mais comum e necessita de automação, para isso podem ser usadas técnicas de visão computacional. Foi usado como base para o desenvolvimento deste trabalho o algoritmo de segmentação NHD, a partir deste algoritmo foi implementado o algoritmo de *tracking*. Para a otimização do *tracking* foi utilizada a técnica de *OpticalFlow* onde a informação dos quadros anteriores são utilizados para tentar prever a nova posição dos objetos. O sistema permite que o *tracking* continue quando um objeto passa da visão de uma câmera para outra. Para efetuar os testes do sistema foi criado um ambiente virtual, permitindo maior flexibilidade nos testes.

Palavras-chave: Sistema de monitoramento. Visão computacional. Processamento de imagens. *Tracking* de objetos. *OpticalFlow*.

ABSTRACT

This paper presents a tracking system capable of following objects over several cameras. The use of surveillance cameras is an increasingly common practice that needs automation, and computational vision techniques can be used to solve that problem. The basis for this paper's development was the NHD segmentation algorithm, upon which the tracking algorithm was built. The tracking procedure was optimized with the OpticalFlow technique, where information from previous frames are used to predict an object's position. The system can track the movement of an object from one camera to another. To perform tests a virtual world was created. This allows more flexibility with test scenarios.

Key-words: Monitoring system. Computional vision. Image processing. *Tracking* of objects. *OpticalFlow*.

LISTA DE ILUSTRAÇÕES

Figura 1– Imagem binária 16
Figura 2 – Comparação dos clusters 19
Quadro 1 – Fórmulas *OpticalFlow* 20
Figura 3 – Controle das *threads* 21
Figura 4 – Cadeia de câmeras 22
Figura 5 – Adaptação de iluminação NHD 23
Figura 6 – Interface DanioTrack 23
Figura 7 – Interface Simi MotionCapture3D 24
Figura 8 – Diagrama de caso de uso 26
Figura 9 – Diagrama de classe 27
Figura 10 – Diagrama de sequência 30
Figura 11 – Fluxograma do *tracking* 32
Quadro 2 – Código do *tracking* 34
Quadro 3 – Código da busca de componentes conexos 35
Quadro 4 – Código gerenciamento das visões 37
Figura 12 – Interface do sistema 38
Figura 13 – Visões do ambiente 39
Figura 14 – Resultado aplicação algoritmo de detecção de movimento 40
Figura 15 – Resultado do tracking 40
Quadro 5 – Resultados tracking 43

LISTA DE SIGLAS

RF – Requisito funcional

RNF – Requisito não funcional

UML - *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO **13**
1.1 OBJETIVOS DO TRABALHO 14
1.2 ESTRUTURA DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA **15**
2.1 DETECÇÃO DE OBJETOS EM MOVIMENTO 15
2.2 CLASSIFICAÇÃO DE OBJETOS 16
2.3 TRACKING COM VÁRIAS CÂMERAS 17
2.3.1 *Tracking* de objetos 18
2.3.1.1 Algoritmo NHD 18
2.3.1.2 Algoritmo de *Optical Flow* 19
2.3.2 Gerenciamento das várias visões 20
2.3.3 Organização das câmeras 21
2.4 TRABALHOS CORRELATOS 22
3 DESENVOLVIMENTO **25**
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 25
3.2 ESPECIFICAÇÃO 25
3.2.1 Diagrama de caso de uso 25
3.2.2 Diagrama de classe 26
3.2.2.1 Classes `Carro` e `Camera` 27
3.2.2.2 Classes `ViewObject` e `GlobalObject` 28
3.2.2.3 Classe `ThreadView` 28
3.2.2.4 Classe `OpticalFlow` 29
3.2.2.5 Classe `GlView` 29
3.2.2.6 Classe `Frames` 29
3.2.3 Diagrama de sequência 30
3.3 IMPLEMENTAÇÃO 31
3.3.1 Técnicas e ferramentas utilizadas 31
3.3.1.1 Processo de *tracking* em cada câmera 32
3.3.1.2 Gerenciamento das várias visões 36
3.3.2 Operacionalidade da implementação 38
3.3.2.1 Visão geral do ambiente e das câmeras 39

3.3.2.2 Visualização da aplicação dos algoritmos 39
3.4 RESULTADOS E DISCUSSÃO 41
3.4.1 Ambiente de testes 41
3.4.2 Testes do processo de *tracking* 42
3.4.3 Posicionamento das câmeras 44
4 CONCLUSÕES 45
4.1 EXTENSÕES 45
REFERÊNCIAS BIBLIOGRÁFICAS 47

1 INTRODUÇÃO

Sistemas de vigilância são amplamente utilizados no cotidiano. As câmeras de monitoramento estão muito presentes no dia a dia. Encontrar um aviso do tipo “você está sendo filmado” pode ser considerada uma situação corriqueira. No Brasil, por exemplo, existem cerca de 1,3 milhões de câmeras de vigilância (PORTAL MS, 2009). Mas qual a serventia destas câmeras se ninguém estiver monitorando suas imagens?

A solução tradicional para o problema é a criação de centrais de vigilância, onde uma pessoa tem a responsabilidade de acompanhar o que está sendo capturado pelas câmeras procurando por situações incomuns. Segundo Garcia (2001, p. 113), seres humanos não são bons em detectar eventos, devido ao fato de não conseguirem manter a concentração em uma atividade durante longos períodos de tempo. Isso implica no aumento da probabilidade de falhas acontecerem no processo de vigilância.

Sendo assim, é possível verificar que existe uma demanda de automatização no processo de monitoramento, objetivando reduzir o número de falhas provenientes da ineficiência de um operador humano. Uma ferramenta que identifique os objetos e suas ações através das imagens capturadas pelas câmeras é um subsidio importante para automatização deste processo.

A construção de uma ferramenta deste tipo pode ser feita utilizando técnicas de visão computacional. De acordo com Parker (1994, p. 1), o objetivo da visão computacional é desenvolver formas de um computador interpretar imagens para alguma aplicação útil. As aplicações variam entre classificação de áreas claras e escuras em imagens e detecção de movimento, até aplicações que envolvem coordenadas 3D precisas e reconhecimento de objetos.

Diante do exposto, propõe-se o desenvolvimento de uma ferramenta que utiliza técnicas de visão computacional para fazer o acompanhamento (*tracking*) de objetos de interesse que serão identificados na cena. O acompanhamento será feito a partir de imagens adquiridas por múltiplas câmeras.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta que faça *tracking* de objetos em uma cena utilizando técnicas de visão computacional.

Os objetivos específicos do trabalho são:

a) fazer a identificação dos objetos em movimento na cena;
b) definir uma técnica para fazer o *tracking* dos objetos;
c) utilizar várias câmeras para efetuar o *tracking*.

1.2 ESTRUTURA DO TRABALHO

No capítulo 2 é apresentada a revisão bibliográfica que foi utilizada para dar suporte ao desenvolvimento do sistema. Neste capítulo são apresentados conceitos e técnicas referentes a detecção e classificação de objetos e *tracking* com várias câmeras.

O capítulo 3 mostra as etapas de desenvolvimento do sistema. Primeiramente são apresentados os requisitos da aplicação, após isso são mostrados os diagramas utilizados para a especificação da ferramenta. Na sequência são descritas as ferramentas e técnicas utilizadas na implementação do sistema e sua operacionalidade. Por fim são apresentados os resultados e discussão.

No capítulo 4 é feita a conclusão do trabalho e são feitas sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 são apresentadas técnicas utilizadas para a detecção de objetos em movimento em uma cena. Na seção 2.2 são apresentadas etapas de um processo de classificação de objetos. Na seção 2.3 são abordados temas relacionados ao *tracking* de objetos utilizando várias câmeras. Na seção 2.4 são descritos três trabalhos correlatos.

2.1 DETECÇÃO DE OBJETOS EM MOVIMENTO

O movimento é uma poderosa maneira que humanos e animais usam para extrair objetos ou regiões de interesse de um fundo com detalhes irrelevantes. Uma das abordagens mais simples para a detecção de movimento entre duas imagens em tempos diferentes é a comparação pixel por pixel (GONZALEZ; WOODS, 2008, p. 778).

Uma forma de fazer isso é através da criação de outra imagem onde os pixels são gerados dependendo da diferença entre as imagens que estão sendo comparadas. Com isso, se os dois pixels comparados apresentarem uma diferença substancial, maior que um limiar definido, o valor do novo pixel será um, em caso contrário, o valor será zero. Supondo que exista uma imagem contendo apenas objetos estáticos, quando esta é comparada com outra subseqüente na mesma cena, onde há um objeto em movimento, o resultado é que as áreas por onde o objeto passou serão marcadas com o valor um e as que permaneceram iguais serão marcadas com o valor zero (GONZALEZ; WOODS, 2008, p. 778).

O processo descrito gera uma imagem binária, formada por apenas duas cores: preto e branco. César Junior e Costa (2001, p. 203) descrevem imagens binárias como o tipo mais simples e útil de imagens, representando uma cena com apenas dois valores, geralmente zero e um. Freqüentemente este tipo de imagens pode ser entendido de forma intuitiva classificando os elementos como objetos, também chamados de *foreground* e *background*.

A Figura 1 mostra uma imagem e a sua versão binária gerada a partir de um processo semelhante ao apresentado.

Fonte: Butler et al. (2005, p. 6).

Figura 1– Imagem binária

Com a imagem neste formato pode ser executado um algoritmo para a detecção dos limites dos objetos identificados. Conforme Parker (1994, p. 75), para a maioria dos objetos as suas bordas definem sua forma com qualidade suficiente para caracterizá-lo.

A partir das operações descritas é possível fazer a identificação de objetos em movimento em uma cena. Com base nos seus contornos pode-se definir uma forma, porém não é identificado que tipo de objeto é e, também, não é possível verificar se ele encontra-se em uma categoria especificada ou não.

2.2 CLASSIFICAÇÃO DE OBJETOS

Com a detecção dos objetos em cena e com a definição de seus contornos é possível partir para um próximo passo, a classificação do objeto.

Para fazer a interpretação dos objetos os sistemas de visão computacional precisam de um conhecimento prévio sobre suas formas. Isso pode ser conseguido através de duas maneiras distintas. A primeira é ter uma base já existente de formas, onde os padrões já estão previamente armazenados. A segunda trata-se do reconhecimento a partir de aprendizado. Neste método, um conjunto de imagens de treinamento é usado para uma fase de aprendizagem, criando assim uma base de formas para futuros reconhecimentos (FACON, 1993, p. 173).

Para exemplificar a etapa de treinamento será apresentada uma das técnicas que podem ser utilizadas para o reconhecimento, as redes neurais artificiais. Esta técnica utiliza treinamento prévio para a criação da base de formas.

“Redes neurais artificiais são sistemas computacionais [...] que imitam as habilidades computacionais do sistema nervoso biológico, usando um grande número de simples

neurônios artificiais interconectados" (LOESCH; SARI, 1996, p. 5).

Redes neurais *feedforward* multicamadas são boas para reconhecimento de padrões como, por exemplo, a rede percepton multicamadas (LOESCH; SARI, 1996, p. 59). O processo de treinamento é apresentado a seguir para exemplificar a fase de aprendizado para o reconhecimento de padrões.

O treinamento deste tipo de rede é descrito por Loesch e Sari (1996, p. 61) como um aprendizado supervisionado. Um conjunto de dados de treinamento deve ser criado e padrões de treinamento definidos. O conjunto de dados de aprendizagem deve conter exemplares de cada categoria que a rede aprenda.

O treinamento utilizado por este tipo de rede é feito a partir de um processo de *feedback*. A rede inicia com valores aleatórios para os seus neurônios e, após a entrada passar pela rede, o resultado é comparado com o esperado, sendo que a diferença entre os dois é usada para ajustar o peso dos neurônios. Após os ajustes dos pesos o processo é repetido com outra entrada. Quando todos os dados forem processados pela rede o processo é reiniciado até que não sejam mais necessários ajustes nos pesos, ou seja, enquanto a rede não apresentar uma capacidade de reconhecimento satisfatória o treinamento é refeito utilizando nos neurônios os pesos da iteração anterior.

Depois da rede treinada, ela está pronta para reconhecer as formas aprendidas na fase de treinamento. Se a rede receber como entrada uma forma que estava no conjunto de dados de treinamento ela é capaz de determinar qual o tipo do objeto de entrada.

2.3 TRACKING COM VÁRIAS CÂMERAS

Tyagi et al. (2007, p. 2) colocou o *tracking* como um caso especial de registro de imagens, onde procura-se um objeto dado em uma imagem alvo. O espaço de busca é limitado devido à suposição que o objeto tem uma trajetória de movimento contínua, e é esperado que ele se encontre nas proximidades da sua localização anterior.

O *tracking* deve ocorrer através de uma cadeia de câmeras, em que um objeto, após ser identificado por uma das câmeras, é acompanhado por essa. No momento que o objeto for para a área de visão de outra câmera o *tracking* continua, reconhecendo ele como sendo o mesmo que estava sendo acompanhado na outra visão.

2.3.1 *Tracking* de objetos

Muitos sistemas de *tracking* utilizam a detecção de movimento como primeiro passo no acompanhamento do objeto. Uma vez que o objeto é localizado, vários métodos podem ser usados para manter o *tracking* (DENMAN; CHANDRAN; SRIDHARAN, 2007).

Um destes métodos é o o*ptical flow*, Denman, Chandran e Sridharan (2007) propuseram um algoritmo que utiliza o o*ptical flow* para manter o *tracking* do objeto baseado no algoritmo de segmentação adaptativa NHD. A integração entre as duas técnicas permite que o *optical flow* seja aplicado somente aos pixels em movimento, reduzindo o consumo de recursos e a presença de erros no *tracking*.

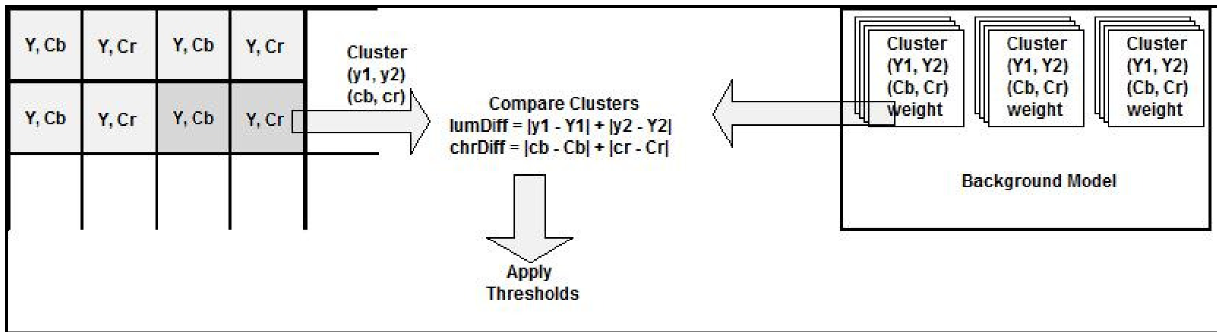
2.3.1.1 Algoritmo NHD

Como primeiro passo para o *tracking* pode-se utilizar um algoritmo de detecção de movimento, como por exemplo, o algoritmo de segmentação adaptativa NHD.

Este algoritmo usa como entrada uma imagem com cores no modelo Y'CbCr e com formato 4:2:2. Neste sistema as cores são representadas pela sua luminância e cromância. O componte Y contém altas freqüências de luminância em escalas de cinza. E os componentes Cb e Cr guardam as altas freqüências das informações de cores.

Para a compactação no formato 4:2:2 os pixels são agrupados em dois, sendo dois pixels de largura por um de altura. O novo pixel é formado por dois valores de luminância, um de cada pixel, e dois valores de cromância, formado pela média dos valores de cromância dos dois pixels (SANTOS, 2008).

Estes grupos de pixels são chamados de *clusters*, onde cada um tem um centróide, formado pelo par de valores de luminância, pelo par de valores de cromância, e um peso. No processo de segmentação os *clusters* da imagem sendo processada são comparados aos clusters no modelo de fundo. O processo de comparação entre os pixels é apresentado na Figura 2.



Fonte: Denman, Chandran e Sridharan (2007).

Figura 2 – Comparação dos *clusters*

Quando um *cluster* é considerado equivalente ao do fundo seus valores de cor e o seu peso são adaptados para que as informações do *cluster* processado sejam ajustadas. Se este pixel não for considerado como pertencente ao fundo então os valores do novo pixel são incorporados ao modelo e este pixel é classificado como em movimento.

Os *clusters* e seus pesos mudam gradualmente enquanto os *frames* são processados, permitindo o sistema se adaptar a mudanças no fundo da cena. Assim novos objetos podem ser incorporados na cena, e depois de algum tempo eles são incorporados ao modelo de fundo (DENMAN; CHANDRAN; SRIDHARAN, 2007).

2.3.1.2 Algoritmo de *Optical Flow*

No algoritmo proposto por Denman, Chandran e Sridharan (2007), que se baseia no algoritmo NHD, os *clusters* detectados como em movimento são o ponto de partida para o processo de *tracking*, evitando que todos os pixels precisem ser analisados.

Quando o movimento é detectado em um pixel, a área a sua volta é examinada para determinar o *optical flow* para aquele pixel. O tamanho da área examinada deve ser determinado pela máxima aceleração permitida para um pixel, tanto no eixo `x` quanto no eixo `y`, estes valores devem ser determinados de acordo com as cenas a serem analisadas. Esta área é analisada de dentro para fora, começando no pixel central e depois continuando nos pixels mais externos até que seja encontrado o pixel correspondente.

Uma vez que o movimento de um pixel foi determinado, sua nova posição pode ser prevista. Com um modelo de velocidade constante a localização do novo pixel `p`, no próximo *frame* é dado pelas fórmulas apresentadas no Quadro 1 (para `x` e `y`).

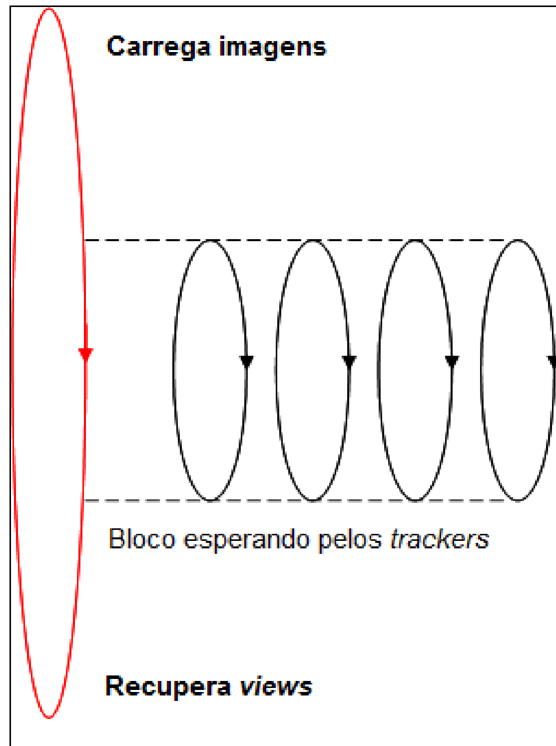
$$p_x^{n+1} = p_x^n + (p_x^n - p_x^{n-1})$$
$$p_y^{n+1} = p_y^n + (p_y^n - p_y^{n-1})$$

Quadro 1 – Fórmulas *OpticalFlow*

Onde p_x^{n-1} e p_y^{n-1} são as posições do pixel `p` no *frame* anterior, p_x^n e p_y^n são as posições de `p` no *frame* atual e p_x^{n+1} e p_y^{n+1} são as posições esperadas do pixel `p` no próximo *frame.* Se o pixel já era classificado como em movimento, então a posição esperada é usada como posição inicial para a busca.

2.3.2 Gerenciamento das várias visões

Para fazer o *tracking* através de várias câmeras, Denman et al. (2006, p. 2 - 3) propõem o uso de um módulo de gerenciamento. Este módulo é responsável por agregar os *tracks* das várias visões do ambiente, devendo determinar quando ele muda de câmera e quando ele saiu da área coberta pelo sistema. Para possibilitar o processamento paralelo no sistema, o que melhora o desempenho, são usadas *threads*. Para cada câmera deve existir uma *thread* que é responsável pelo *tracking* dos objetos capturados pela câmera associada a ela. Além das *threads* responsáveis por cada câmera é criada outra para o módulo de gerenciamento. Este módulo fornece às *threads* responsáveis pelo *tracking* as imagens das câmeras, e busca os *tracks* ao final de cada *frame*. O processo de gerenciamento das *threads* é apresentado na Figura 3.



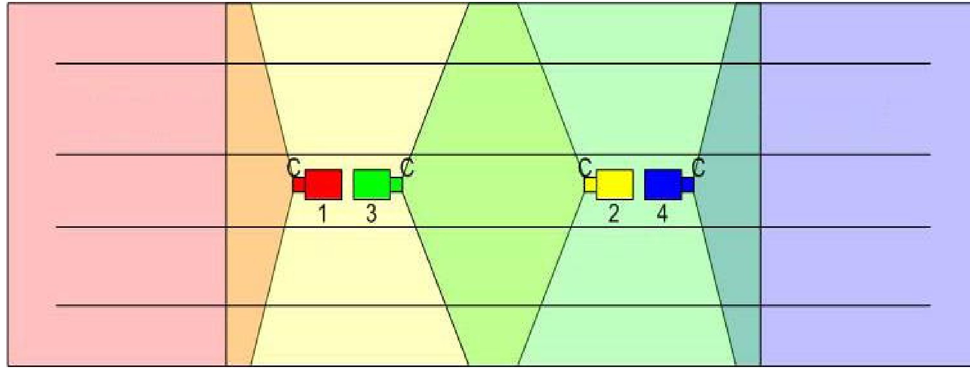
Fonte: Denman et al. (2006, p. 3).

Figura 3 – Controle das *threads*

Os objetos que são acompanhados pelo sistema são representados como *Tracked objects*, onde cada um deles consiste em um ou mais *objects views*. Um *object view* representa um objeto identificado por uma câmera, ele armazena informações referentes à localização do objeto na imagem. O *tracked object* armazena também a posição do objeto usando coordenadas comuns aos vários *trackers* (as *threads* responsáveis pelo acompanhamento de uma *view*).

2.3.3 Organização das câmeras

No sistema proposto por Denman et al. (2006, p. 4), são usadas quatro câmeras calibradas colocadas conforme apresentado na Figura 4.



Fonte: Denman et al. (2006, p. 4).

Figura 4 – Cadeia de câmeras

Cada uma das câmeras tem uma área de transição com a próxima câmera, nesta área existe uma sobreposição dos seus campos de visão. Quando um *track* entra nesta área de transição, o módulo de gerenciamento cria um falso *track* na próxima câmera. Quando o próximo *frame* é processado, o *tracker* desta câmera tenta combinar o falso *track* com um objeto detectado, se ele conseguir o *tracking* continua na próxima *view*.

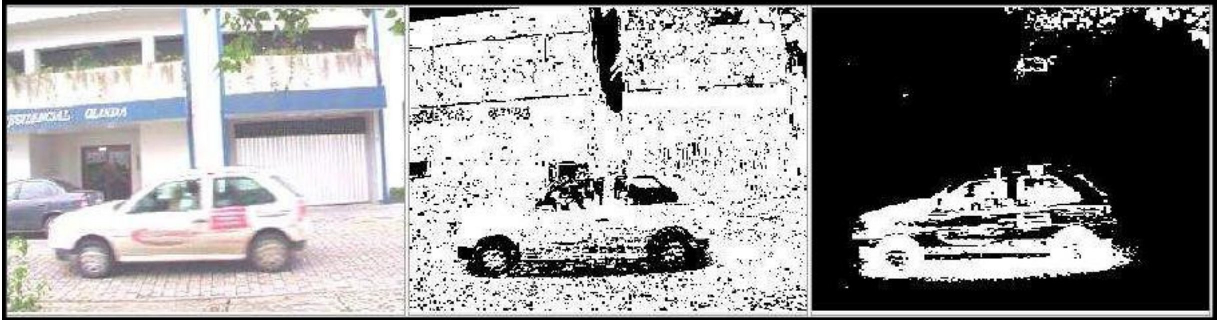
Com toda a estrutura de câmeras montada e devidamente configurada, a aplicação responsável pelo *tracking* pode ser iniciada e os objetos na cena podem começar a ser monitorados.

2.4 TRABALHOS CORRELATOS

Foi usada como base para o desenvolvimento do trabalho proposto a ferramenta descrita em Santos (2008), que efetua a contagem de veículo utilizando técnicas de visão computacional. Existem algumas ferramentas comerciais que utilizam técnicas de *tracking*, sendo que dentre elas foram selecionadas: DanioTrack (QUBIT SYSTEMS, 2010) e o Simi MotionCapture3D (SIMI REALITY MOTION SYSTEMS, 2010).

Para a criação de um sistema de contagem de veículos Santos (2008) utilizou um conjunto de técnicas que permitia a identificação de um veículo independente de cenário e iluminação. Foi utilizado o algoritmo de segmentação adaptativa NHD para diferenciação de fundo e dos objetos em movimento. Para descrever as formas dos objetos são utilizados descritores de Fourier. Já para a classificação do objeto identificado como sendo um veículo foi utilizada uma rede neural artificial. O algoritmo utilizado para a segmentação apresenta como grande vantagem a independência de cenário e iluminação. A Figura 5 demonstra a adaptação do algoritmo NHD a diferenças de luminosidade em relação ao algoritmo de

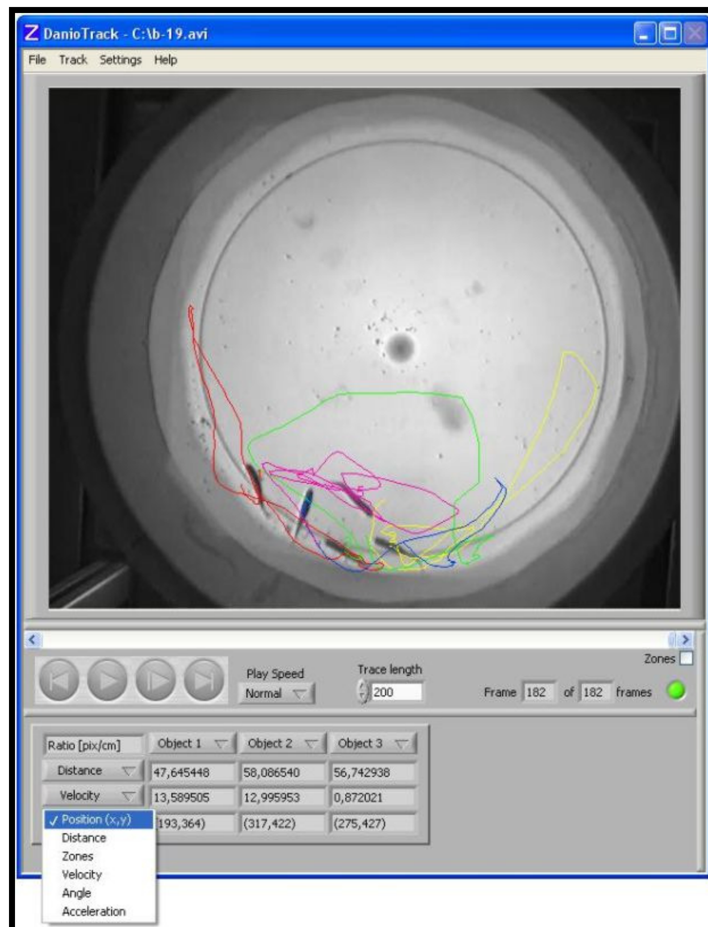
remoção de fundo.



Fonte: Santos (2008).

Figura 5 – Adaptação de iluminação NHD

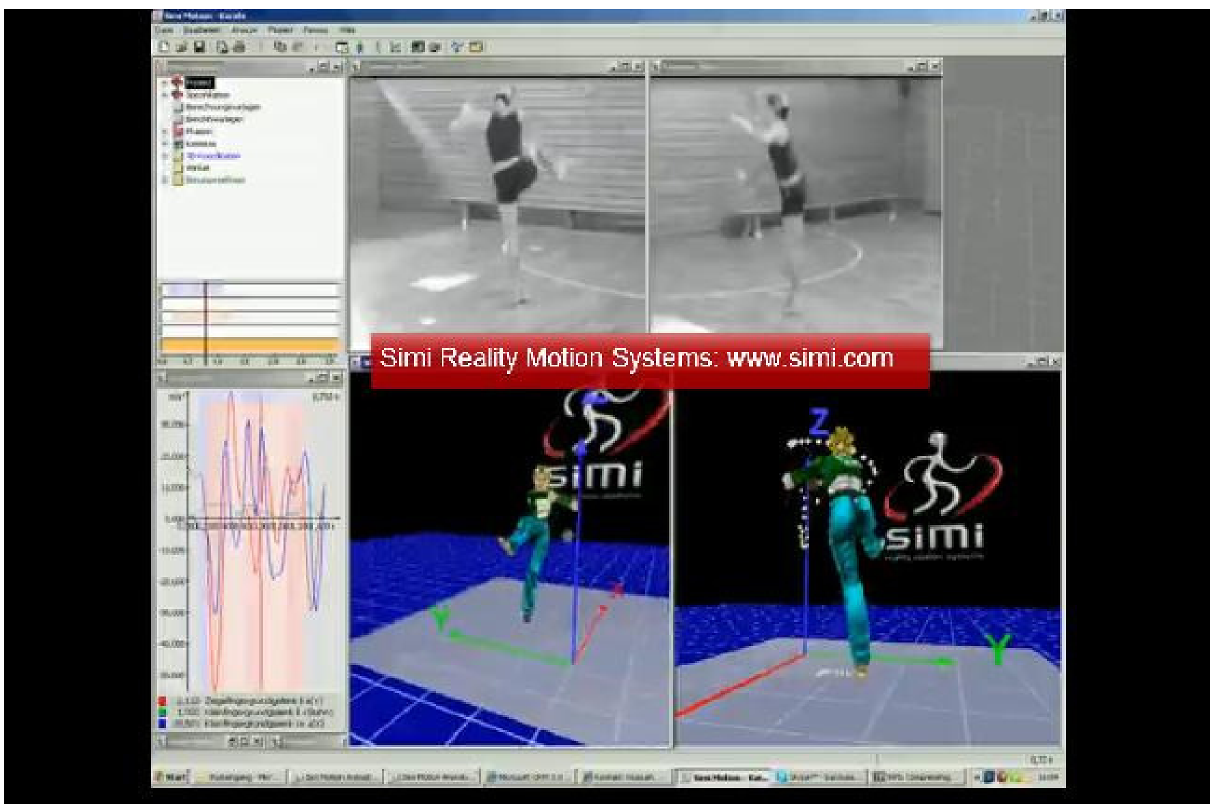
Qubit Systems (2010) afirma que a ferramenta DanioTrack permite acompanhar o comportamento de vários animais com base em arquivos de vídeo sem a utilização de marcações especiais. O usuário pode definir áreas de interesse e indesejáveis na cena, para restringir a zona de atuação do *tracking*. O sistema pode calcular a distância percorrida, velocidade (atual, máxima e média), aceleração (atual e máxima), ângulos e mudanças de direção. Após o *tracking* é gerado um arquivo com a tabulação dos dados identificados durante o monitoramento dos animais. A Figura 6 mostra a interface da ferramenta.



Fonte: Qubit Systems (2010).

Figura 6 – Interface DanioTrack

O Simi MotionCapture3D (SIMI REALITY MOTION SYSTEMS, 2010) é uma ferramenta a qual permite a captura de movimentos a partir de imagens. Pode ser utilizada para a captura de movimento de vários tipos de objetos, como humanos, animais e máquinas em um espaço 3D. A ferramenta tem diversas aplicações, como seqüências animadas por computador, vídeo games, desenhos animados e simulações. Por ser um sistema óptico de captura de movimentos não se fazem necessários sensores com custo mais elevado, apenas é preciso câmeras e marcações especiais nas articulações dos objetos, permitindo uma grande liberdade de movimento aos participantes da cena. Os dados extraídos podem ser exportados para diversos formatos de arquivos de ferramentas comerciais. A Figura 7 apresenta a ferramenta em execução.



Fonte: Simi Reality Motion Systems (2010).

Figura 7 – Interface Simi MotionCapture3D

3 DESENVOLVIMENTO

Neste capítulo são abordadas as atividades relacionadas ao projeto e desenvolvimento do sistema proposto neste trabalho.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta deve:

a) disponibilizar uma interface para a visualização de um cenário a partir de imagens de várias câmeras (Requisito Funcional - RF);
b) identificar objetos em movimento na cena (RF);
c) realizar o *tracking* dos objetos (RF);
d) adquirir os dados da cena de imagens provenientes de várias câmeras (RF);
e) ser desenvolvido utilizando programação orientada a objetos (Requisito Não Funcional – RNF);
f) ser implementado utilizando a linguagem de programação C++ (RNF);
g) ser implementado utilizando o ambiente de programação Eclipse Galileo (RNF).

3.2 ESPECIFICAÇÃO

O sistema apresentado utiliza alguns diagramas da *Unified Modeling Language* (UML). Foi utilizada a ferramenta JUDE Community para o desenvolvimento dos diagramas de caso de uso, de classe e de sequência.

3.2.1 Diagrama de caso de uso

A Figura 8, apresenta o caso de uso disponível para o usuário do sistema.

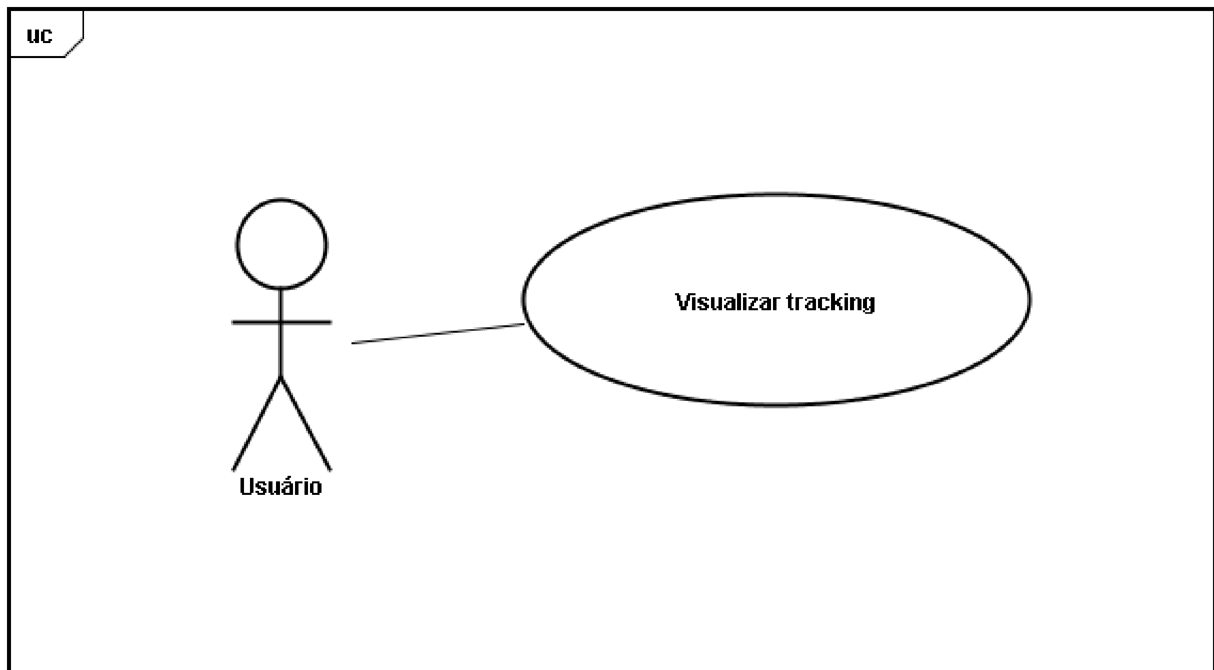


Figura 8 – Diagrama de caso de uso

No caso de uso `Visualizar tracking`, o usuário irá visualizar a aplicação dos algoritmos de segmentação, bem como os rastros identificados pelo *tracking* dos objetos que estão sendo capturados pelas câmeras do sistema. O caso de uso começa com o usuário selecionando uma das simulações disponíveis, então o sistema faz a simulação, aplica os algoritmos e os apresenta ao usuário.

3.2.2 Diagrama de classes

A Figura 9 apresenta o diagrama de classes especificado para a aplicação. Nele se encontram as classes que fazem a simulação do ambiente, a execução dos algoritmos e também a apresentação dos resultados ao usuário.

Também são apresentadas classes que representam as câmeras e carros da simulação, e os objetos identificados pelo algoritmo tanto de forma local, em cada câmera, como global.

A classe `Point2D` serve para representar um ponto no sistema de coordenada 2D e é usado por algumas classes do sistema.

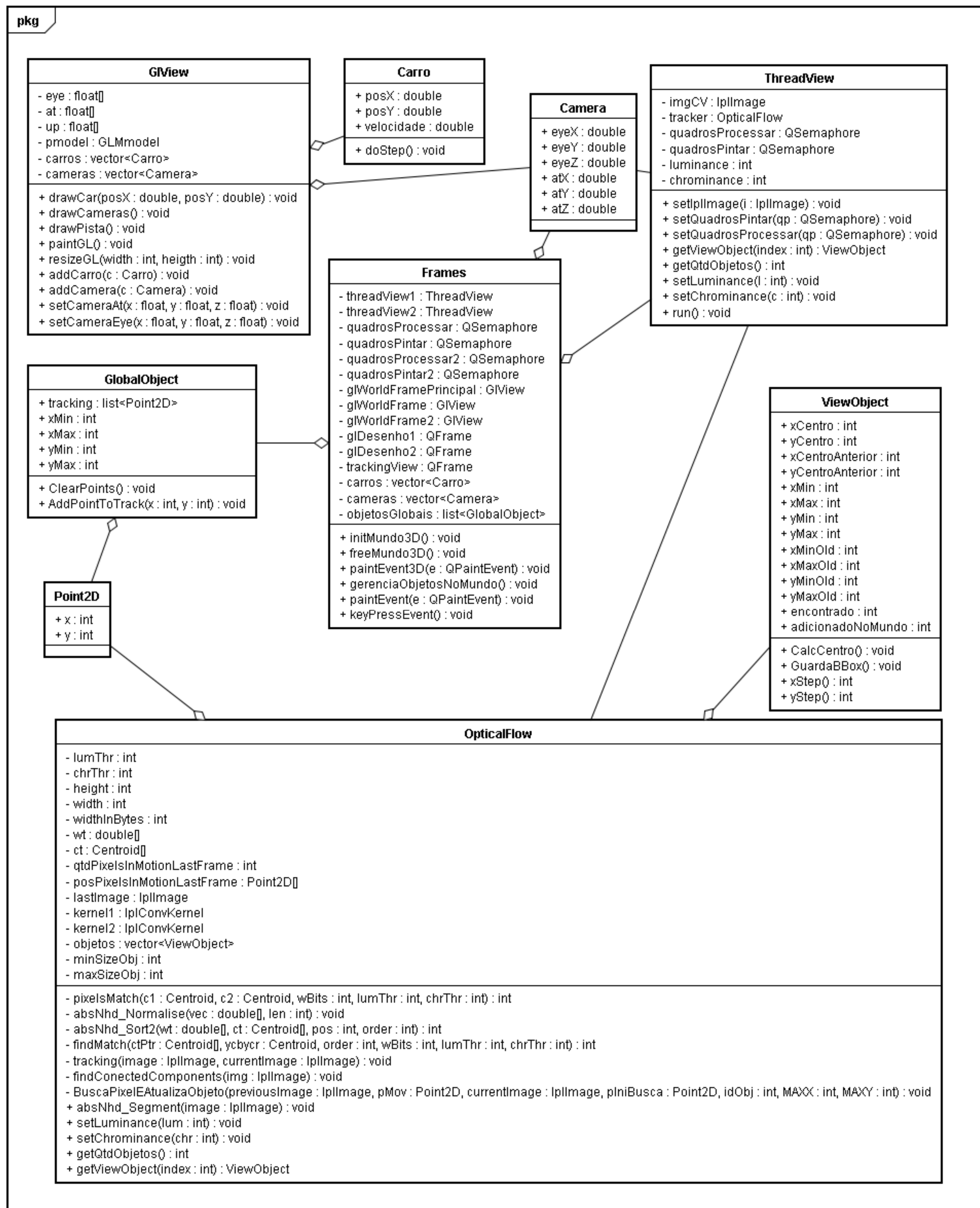


Figura 9 – Diagrama de classe

3.2.2.1 Classes `Carro` e `Camera`

Para representar os objetos simulados são usadas as classes `Carro` e `Camera`. A classe `Carro` armazena a posição e a velocidade do carro que está sendo simulado. E o método

`doStep` aplica a simulação mudando a posição de acordo com a velocidade do objeto.

Para a representação das visões da simulação é utilizada a classe `Camera`, nela são armazenadas duas coordenadas no espaço 3D, uma onde fica a câmera simulada e outra para que ponto ela está olhando.

3.2.2.2 Classes `ViewObject` e `GlobalObject`

As classes `ViewObject` e `GlobalObject` representam os objetos no mundo real, sendo que a primeira em relação a uma câmera, e o segundo referente ao contexto geral da cena. Nestas classes são armazenadas informações referentes aos limites dos objetos, com essas informações é possível determinar a posição do objeto.

Na classe `ViewObject` também são armazenadas informações das coordenadas anteriores do objeto (essa informação é utilizada para se calcular o *OpticalFlow* do objeto). A execução dos cálculos dos valores usados pelo *OpticalFlow*, são feitas pelos métodos `xStep` e `yStep`.

Na classe `GlobalObject` é onde que o *tracking* do objeto realmente é armazenado, a classe tem uma lista que armazena todos os pontos centrais (calculados a partir dos limites do objeto) por onde o objeto passou, determinando sua trajetória.

3.2.2.3 Classe `ThreadView`

A classe `TheadView` é responsável por executar o *tracking* para uma câmera. Esta classe é um *thread* que permite o processamento paralelo das vários *views* do cenário. Ela basicamente recebe um *frame* para processar, executa os algoritmos sobre este *frame* e retorna o *frame*.

Para efetuar o processo de recepção e entrega de *frames* são utilizadas estruturas do tipo Semáforo, permitindo a sincronização entre o processo principal e as várias *threads* responsáveis pelo processamento.

3.2.2.4 Classe `OpticalFlow`

Na classe `OpticalFlow` ficam as rotinas responsáveis pelo *tracking*, ela é executada pela classe `ThreadView` para o processamento do *frame* recebido. Essa classe foi construída a partir da classe `Segment` onde foi implementado o algoritmo NHD por Santos (2008).

Nesta classe ficam as implementações referentes à detecção de movimento, no método `absNhd_Segment`, detecção de componentes conexos, no método `findConectedComponents`, e de acompanhamento dos objetos implementado no método `tracking`.

3.2.2.5 Classe `GlView`

A classe `GlView` é a responsável por mostrar ao usuário uma visão do ambiente utilizado para fazer os testes da aplicação. Essa classe apresenta ao usuário uma representação dos veículos sendo simulados a partir de um ponto de vista, ou câmera.

Para que os carros simulados sejam visualizados é necessário adicionar uma referência ao um objeto da classe `Carro`, essa referência é adicionada utilizando-se o método `addCarro`.Também é possível adicionar representações das câmeras simuladas, para isto deve-se utilizar o método `addCamera`.

Para configurar qual o ponto de vista do usuário em relação ao ambiente são usados os métodos `setCameraEye` e `setCameraAt`, o primeiro indica em que ponto deverá se localizar a câmera e o segundo indica para onde a câmera estará apontando.

3.2.2.6 Classe `Frames`

A classe `Frames` é responsável pela apresentação ao usuário da simulação do ambiente e aplicação dos algoritmos. Além disso, esta classe é responsável pelo gerenciamento das visões do sistema.

Para a apresentação da simulação do ambiente, são usadas 3 instâncias da classe `GlView`. Sendo uma mostrando a visão geral do ambiente e outras duas para a apresentação das visões das câmeras representadas na simulação. Nestas classes são adicionadas referências aos objetos da classe `Carro`, instanciadas na classe `Frames`, onde também é feita a simulação

que faz com que os veículos se movimentem em todas as visões.

A aplicação dos algoritmos responsáveis pelo *tracking* em cada visão é feita utilizando-se os objetos da classe `ThreadView`, sendo um para cada câmera, através destes objetos a classe `Frames` acessa os valores dos pontos da execução do *tracking*. Depois do processamento ter sido feito, o método `gerenciaObjetosNoMundo` busca a localização dos objetos em cada câmera para objetos com localização global.

Após todo o processamento a tela é atualizada com as novas informações da simulação e dos algoritmos.

3.2.3 Diagrama de sequência

A Figura 10 mostra o diagrama de sequência que permite visualizar o caso de uso `visualizar tracking`.

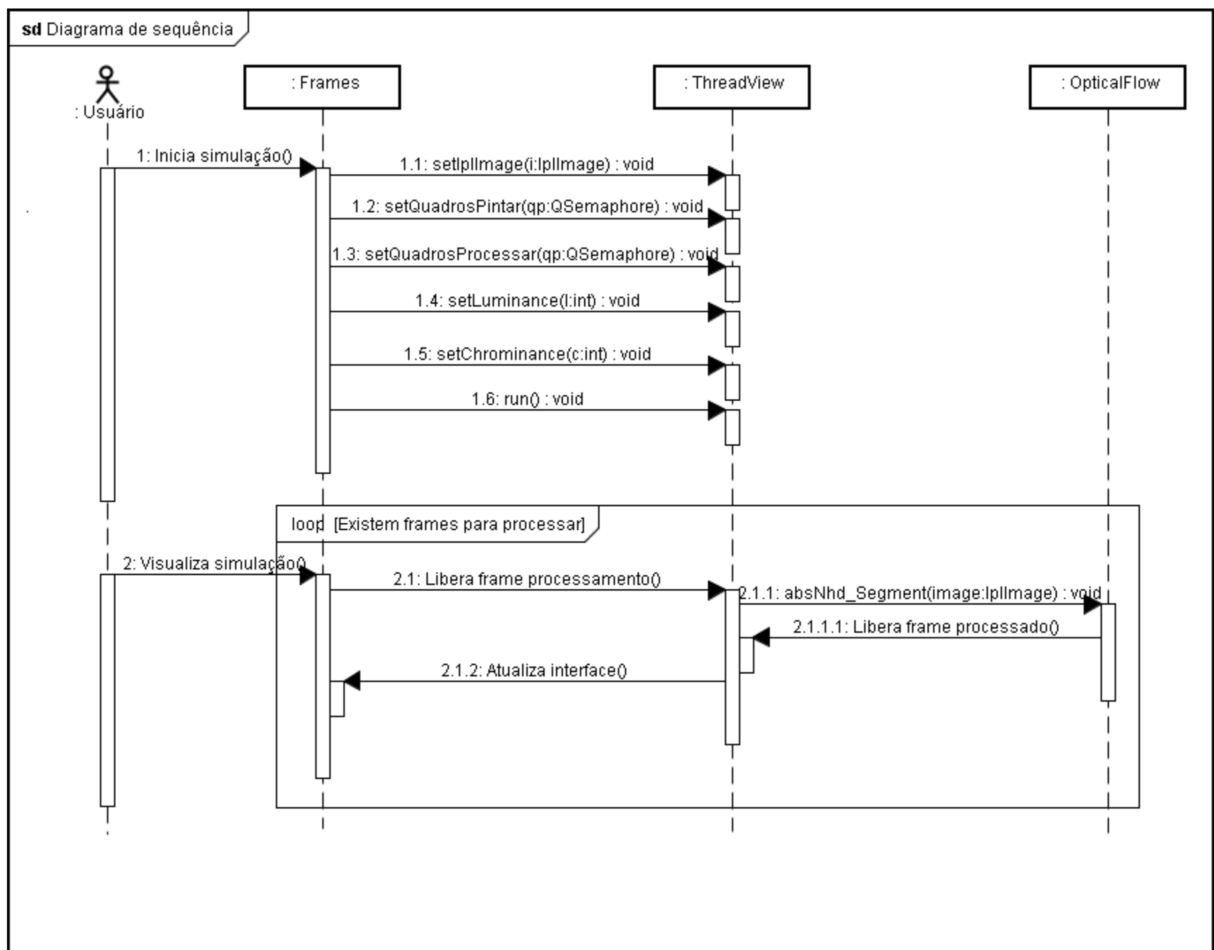


Figura 10 – Diagrama de sequência

O processo é iniciado configurando objetos comuns as classes `Frames` e `ThreadView`

que serão utilizados posteriormente para o processo de envio das imagens e de liberação de *frames* para o processamento e para a atualização deste *frame* na tela. Além disto, são configurados parâmetros utilizados pelo algoritmo de segmentação. Após a configuração inicial ter sido feita é chamado o método `run` da classe `ThreadView` que inicia o processo de visualização.

O processo de visualização é repetido enquanto existirem *frames* a serem processados. Primeiramente a classe `Frames` libera um quadro para processamento, após isso a classe `ThreadView` chama o método `absNhd_Segment` que é responsável por todo o processamento do *tracking*. Quando processamento do *frame* termina o quadro é liberado para ser desenhado na tela e a interface é atualizada.

3.3 IMPLEMENTAÇÃO

A seguir são apresentadas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento do sistema foi utilizado o ambiente de desenvolvimento Eclipse Galileo build 20100218-1602 com o *plug-in* C/C++ Development Tools versão 1.2.2.20100209-1511. Para a criação da interface com o usuário e para o controle de *threads* da aplicação foi utilizada a biblioteca QT 4.6.2, uma das grandes vantagens desta biblioteca e ser multiplataforma. Para integrar o QT ao ambiente de desenvolvimento foi usado o *plug-in* Qt Eclipse integration na versão 1.6.1.

A biblioteca OpenCV versão 1.0.0.1 foi utilizada para os algoritmos de erosão, dilatação e detecção de componentes conexas . Para o desenvolvimento de ambiente de teste do sistema foi utilizada a biblioteca gráfica OpenGL 1.4.

3.3.1.1 Processo de *tracking* em cada câmera

O procedimento de *tracking* inicialmente e feito em cada câmera para depois ser tratado de maneira global. A Figura 11 apresenta um fluxo macro do processo de *tracking*.

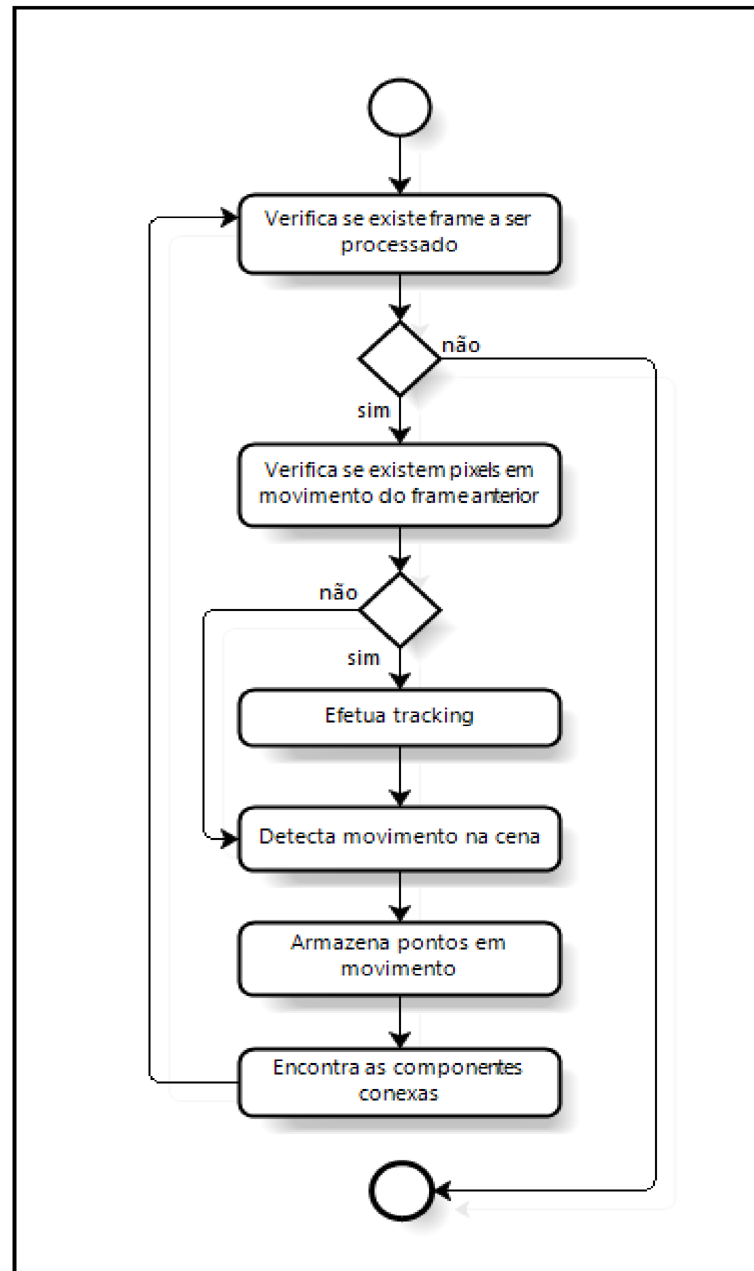


Figura 11 – Fluxograma do *tracking*

Para que seja possível o *tracking* é necessária a separação entre o fundo da cena e os objetos de interesse, neste passo foi usado o algoritmo NHD descrito na secção 2.3.1.1, que foi implementado por Santos (2008). O NHD proposto em Santos (2008) foi modificado para que os pontos que estão em movimento sejam armazenados para uso posterior durante o

tracking. O armazenamento dos pontos busca reduzir o processamento do *tracking* já que não será necessário processar todos os pixels do *frame*, conforme descrito na seção 2.3.1.2.

Após o processo de detecção de movimento ter encontrado algum objeto em movimento na cena é possível efetuar o processo de *tracking*. O Quadro 2 apresenta o código responsável pelo *tracking*.

O processo começa com a inicialização dos objetos (linhas 1 a 10), cada objeto recebe valores mínimos para os seus limites superiores e valores máximos para os seus limites inferiores, permitindo assim que sejam armazenados durante o processo de *tracking* os valores máximos e mínimos de x e y. Após os objetos terem sido inicializados, a partir da linha 11, o algoritmo passa por todos os pixels em movimento que foram armazenados durante a execução do algoritmo de segmentação. A implementação do NHD trata as posições dos pixels em bytes por isso os valores guardados também estão em bytes, porém como a posição dos objetos e tratada em pixels é necessário fazer conversão mostrada nas linhas 18 e 19, onde a posição em bytes é dividida por três já que cada pixel é representado por 3 bytes.

Neste momento o algoritmo passa pela lista de todos os objetos existentes e verifica se este pixel pertence a algum destes objetos (linhas 23 a 26), essa verificação é feita a partir das informações do objeto no *frame* anterior, já que durante este algoritmo essas posições são constantemente alteradas para indicar a nova posição do objeto. Quando um pixel é reconhecido como pertencendo a um objeto, o ponto inicial de busca da nova posição deste pixel é calculado em relação ao movimento feito pelo objeto nos *frames* anteriores (linhas 28 e 29), conforme técnica apresentada na secção 2.3.1.2. Na linha 28 é feita a conversão de pixel para bytes, porém como o algoritmo NHD trabalha com *cluster* formados por dois pixels é necessário que a multiplicação seja feita por 6 e não por 3. Neste momento é feita uma busca para encontrar a posição deste pixel no *frame* atual e atualizar os limites do objeto (linha 37), conforme técnica descrita na secção 2.3.1.2.

```
1.     for ( int i = 0; i < objetos.size(); ++i )
2.     {
3.         ViewObject * vo = objetos[i];
4.         vo->GuardaBBox();
5.         vo->encontrado = false;
6.         vo->xMax = -1;
7.         vo->xMin = width << 4;
8.         vo->yMax = -1;
9.         vo->yMin = height << 1;
10.    }
11.    for (int i = 0; i < qtdPixelsInMotionLastFrame; ++i)
12.    {
13.        pMov = posPixelsInMotionLastFrame[i];
14.        int xpixel, ypixel;
15.        Point pIniBusca;
16.        pIniBusca.x = pMov.x;
17.        pIniBusca.y = pMov.y;
18.        xpixel = pMov.x / 3;
19.        ypixel = pMov.y;
20.        idObj = -1;
21.        for ( int o = 0; o < objetos.size(); ++o )
22.        {
23.            if (xpixel >= objetos[o]->xMinOld
24.                && xpixel <= objetos[o]->xMaxOld
25.                && ypixel >= objetos[o]->yMinOld
26.                && ypixel <= objetos[o]->yMaxOld )
27.            {
28.                pIniBusca.x += objetos[o]->xStep() * 6;
29.                pIniBusca.y += objetos[o]->yStep();
30.                objetos[o]->encontrado = true;
31.                idObj = o;
32.                break;
33.            }
34.        }
35.        if (idObj == -1)
36.            continue;
37.        BuscaPixelEAtualizaObjeto(previousImage, pMov, currentImage,
38.                                  pIniBusca, idObj, MAXX, MAXY);
39.    }
40.    vector<ViewObject*>::iterator itVO;
41.    for ( itVO = objetos.begin(); itVO != objetos.end(); ++itVO)
42.    {
43.        ViewObject * vo = *itVO;
44.        vo->xMin /= 3;
45.        vo->xMax /= 3;
46.        vo->CalcCentro();
47.        if (!vo->encontrado)
48.        {
49.            itVO = objetos.erase(itVO);
50.            --itVO;
51.        }
52.    }
```

Quadro 2 – Código do *tracking*

Depois de todos os pixels armazenados terem sido percorridos (41 a 52), o algoritmo ajusta os pixels e calcula o centro dos objetos além de excluir os objetos que não tiveram nenhum pixel encontrado como pertencente a ele.

Para que seja possível acompanhar vários objetos em uma cena os pixels são

agrupados em componentes conexos. Para que a busca de componentes conexas tenha mais exatidão, são aplicados algoritmos de pós-processamento em cada frame processado pelo NHD. Para estas tarefas foi utilizada a biblioteca OpenCV, o Quadro 3 mostra o código que faz a busca das componentes conexas.

```
1.    cvDilate( img, img, kernel1, 1 );
2.    cvErode( img, img, kernel1, 1 );
3.    cvErode( img, img, kernel2, 1 );
4.    cvDilate( img, img, kernel2, 1 );
5.    ...
6.    cvFindContours(img2, storage, &contour, sizeof(CvContour));
7.    bool achou = false;
8.    while (contour)
9.    {
10.   CvRect screenRect = cvBoundingRect(contour);
11.   ...
12.   achou = false;
13.   for ( int i = 0; i < objetos.size(); ++i )
14.   {
15.     if ((abs(screenRect.height - (objetos[i]->yMax -
16.     objetos[i]->yMin)) < DIFF && abs(screenRect.y - objetos[i]-
17.     > yMin ) < DIFF ) &&
18.     (abs(screenRect.width - (objetos[i]->xMax - objetos[i]->xMin
19.     )) < DIFF && abs(screenRect.x - objetos[i]->xMin) < DIFF))
20.     {
21.         objetos[i]->xMin = screenRect.x;
22.         objetos[i]->xMax = screenRect.x + screenRect.width;
23.         objetos[i]->yMin = screenRect.y;
24.         objetos[i]->yMax = screenRect.y + screenRect.height;
25.         achou = true;
26.         break;
27.     }
28.   }
29.   if (!achou)
30.   {
31.        ViewObject* vo = new ViewObject();
32.        vo->xMin = screenRect.x;
33.        vo->xMax = screenRect.x + screenRect.width;
34.        vo->yMin = screenRect.y;
35.        vo->yMax = screenRect.y + screenRect.height;
36.        vo->CalcCentro();
37.        objetos.push_back(vo);
38.   }
39.   contour = contour->h_next;
40.   }
```

Quadro 3 – Código da busca de componentes conexos

Nas linhas de 1 até 4 são chamadas as funções de erosão e dilatação da biblioteca OpenCV, para fazer o pós-processamento da imagem. Na linha 6 é chamada outra função da biblioteca que faz a busca dos componentes conexos na imagem.

A partir da linha 8 o algoritmo passa por cada uma das componentes conexas. O primeiro procedimento a ser feito é verificar se já existe um objeto na lista que está na posição da componente conexa encontrada. A verificação é feita nas linhas de 15 a 19, nela é utilizado um parâmetro que define uma pequena margem de erro para evitar falhas na comparação.

Caso seja encontrado um objeto que combine com a componente conexa os valores do objeto são atualizados conforme a posição desta (linhas 21 a 26). No caso de não ser encontrado na lista um objeto que seja equivalente a componente conexa um novo objeto é adicionado a lista (linhas 29 a 38).

3.3.1.2 Gerenciamento das várias visões

Depois do processamento em cada câmera do *tracking* é feito um processo global. Nesse processo global as informações obtidas em cada uma das visões e convertida em uma informação global. Dessa forma pode-se identificar quando um objeto em uma câmera é o mesmo que em outra. O Quadro 4 apresenta o código que faz esse gerenciamento.

O algoritmo inicia percorrendo todos os objetos globais conhecidos (linhas 3 e 4). Para cada objeto global ao sistema executa uma série de procedimentos. Primeiramente, para cada câmera, todos os objetos das visões são percorridos, como pode ser visto a partir da linha 9. Na linha 12 pode-se ver a chamada da função para transformar as coordenadas da câmera para as coordenadas globais. Após ter sido feita esta transformação (linhas 13 a 16) são comparadas as posições do objeto global atual e do objeto ajustado. No caso dos objetos serem considerados como sendo o mesmo objeto, são armazenadas as informações atualizadas do objeto (linhas 19 a 24 e linhas 36 a 40).

Se um objeto global não for encontrado em nenhuma visão, então ele é retirado da lista de objetos (linhas 29 a 33). Os objetos locais que não tiveram nenhum objeto encontrado na lista de objetos globais são adicionados à lista (linhas 44 a 59).

```
1.    ViewObject objAjustado;
2.    list<GlobalObject*>::iterator itGO;
3.    for ( itGO = objetosGlobais->begin();
4.    itGO != objetosGlobais->end(); ++itGO)
5.    {
6.        GlobalObject* go = *itGO;
7.        go->encontrado = false;
8.        // repete para cada camera
9.        for ( int j = 0; j < threadView1->getQtdObjetos(); ++j )
10.       {
11.           ViewObject * o = threadView1->getViewObject(j);
12.           ajustaObjeto(objAjustado, AJUSTACAM1, o);
13.           if ( ( abs ( objAjustado.xMin - go->xMin ) < DIFF ||
14.           abs ( objAjustado.xMax - go->xMax ) < DIFF ) &&
15.           (abs ( objAjustado.yMin - go->yMin ) < DIFF ||
16.           abs( objAjustado.yMax - go->yMax ) < DIFF ) )
17.           {
18.               go->encontrado = true;
19.               xMinAux = objAjustado.xMin;
20.               xMaxAux = objAjustado.xMax;
21.               yMinAux = objAjustado.yMin;
22.               yMaxAux = objAjustado.yMax;
23.               xAux = objAjustado.xCentro;
24.               yAux = objAjustado.yCentro;
25.               o->adicionadoNoMundo = true;
26.               break;
27.           }
28.       }
29.       if ( ! go->encontrado )
30.       {
31.           itGO = objetosGlobais->erase(itGO);
32.           itGO--;
33.       }
34.       else
35.       {
36.           go->xMin = xMinAux;
37.           go->xMax = xMaxAux;
38.           go->yMin = yMinAux;
39.           go->yMax = yMaxAux;
40.           go->AddPointToTrack(xAux, yAux);
41.       }
42.   }
43.   // repete para cada camera
44.   for ( int j = 0; j < threadView1->getQtdObjetos(); ++j )
45.   {
46.       ViewObject *o = threadView1->getViewObject(j);
47.       ajustaObjeto(objAjustado, AJUSTACAM1, o);
48.       if (!o->adicionadoNoMundo) {
49.           GlobalObject *go = new GlobalObject();
50.           go->ClearPoints();
51.           go->xMin = objAjustado.xMin;
52.           go->xMax = objAjustado.xMax;
53.           go->yMin = objAjustado.yMin;
54.           go->yMax = objAjustado.yMax;
55.           go->AddPointToTrack(objAjustado.xCentro,
56.           objAjustado.yCentro);
57.           objetosGlobais->push_back(go);
58.       }
59.    }
```

Quadro 4 – Código gerenciamento das visões

3.3.2 Operacionalidade da implementação

Neste seção é apresentada a operacionalidade da implementação da ferramenta de *tracking*. Para demonstrar a aplicação dos algoritmos implementados foi criado um ambiente de teste que simula uma estrada com veículos passando apresentados de vários pontos de vista. Além dessas visões a interface do sistema também apresenta o resultado dos algoritmos. A Figura 12 apresenta a interface do sistema.

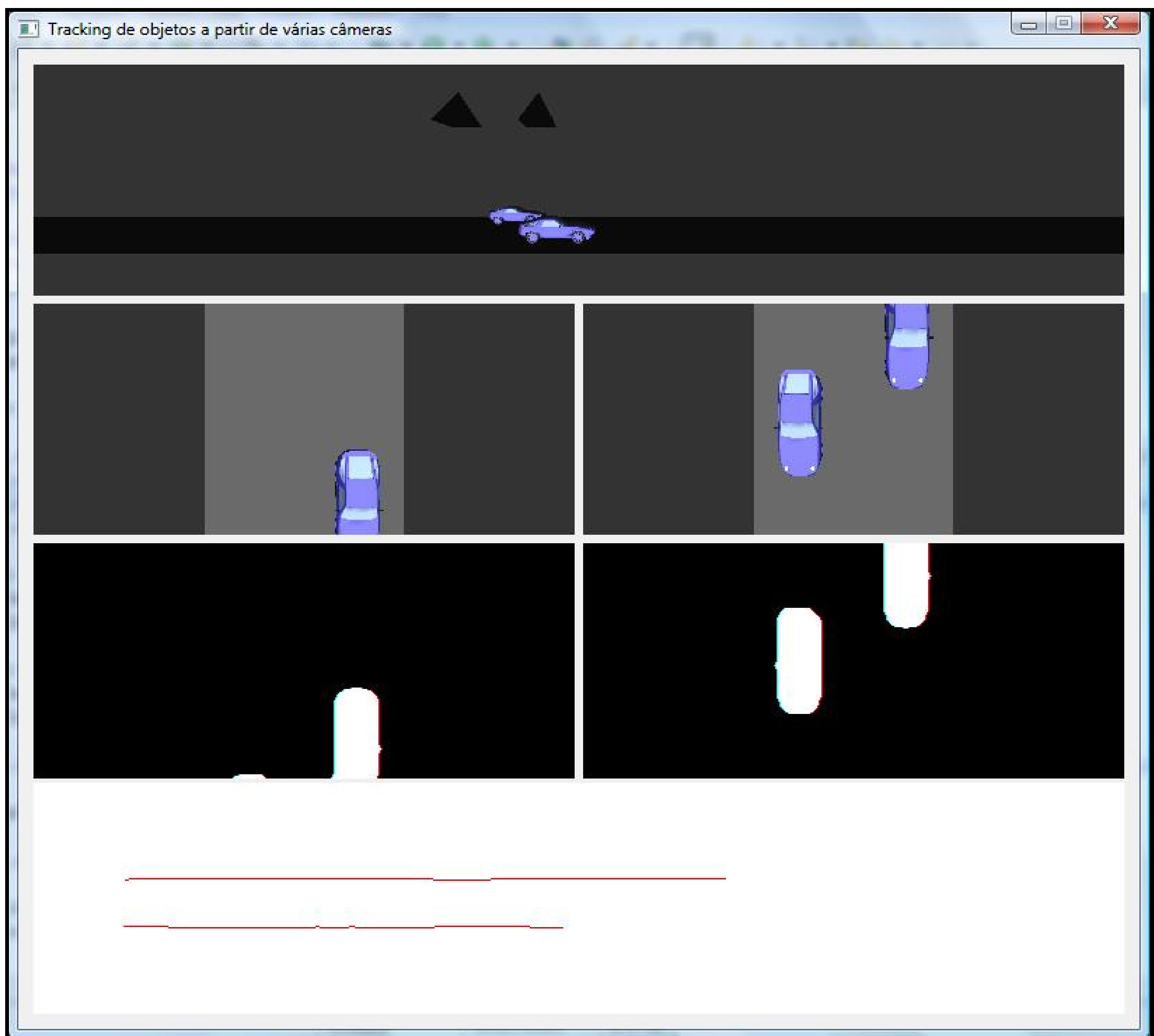


Figura 12 – Interface do sistema

A iteração com o sistema é feita a partir das teclas do teclado. O sistema apresenta algumas possibilidades de simulação, sendo essas iniciadas pelas teclas numéricas de 1 até 4, por exemplo, pressionando a tecla 1 a simulação número 1 irá ser mostrada.

Nas próximas seções são apresentadas de forma mais detalhada as partes da interface.

3.3.2.1 Visão geral do ambiente e das câmeras

O sistema apresenta três visões do ambiente de teste, sendo uma visão geral e outras duas representado as câmeras que capturam as imagens que serão processadas pelo sistema. A Figura 13 apresenta um exemplo dessas representações, as regiões e objetos na tela foram realçados para facilitar a explicação.

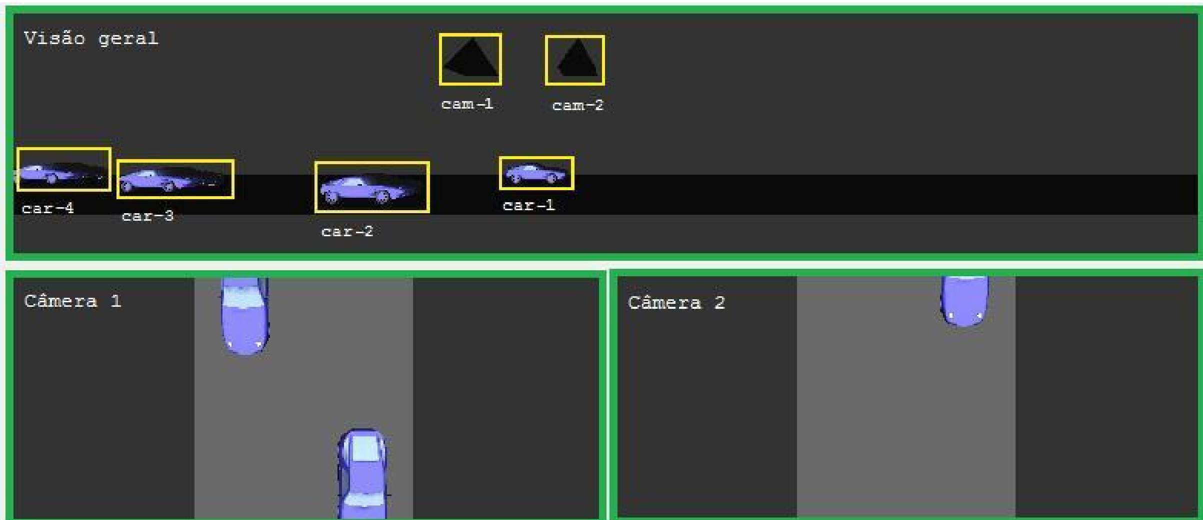


Figura 13 – Visões do ambiente

Na visão geral é possível visualizar a estrada, os carros que estão passando e também as câmeras que monitoram a estrada. No exemplo é possível visualizar 4 carros, porém destes só dois estão sendo monitorados pelas câmeras. Na câmera 1, representada na visão geral por `cam-1`, é possível visualizar os veículos representados por `car-1` e `car-2` já na outra câmera é possível visualizar apenas uma parte de `car-1`.

3.3.2.2 Visualização da aplicação dos algoritmos

Na tela do sistema são apresentados ao usuário o resultado da detecção de movimento para cada uma das visões e o caminho identificado pelo *tracking* dos carros. A Figura 14 mostra o resultado da aplicação do algoritmo de detecção de movimento e a Figura 15 mostra o resultado da aplicação do algoritmo de *tracking*.

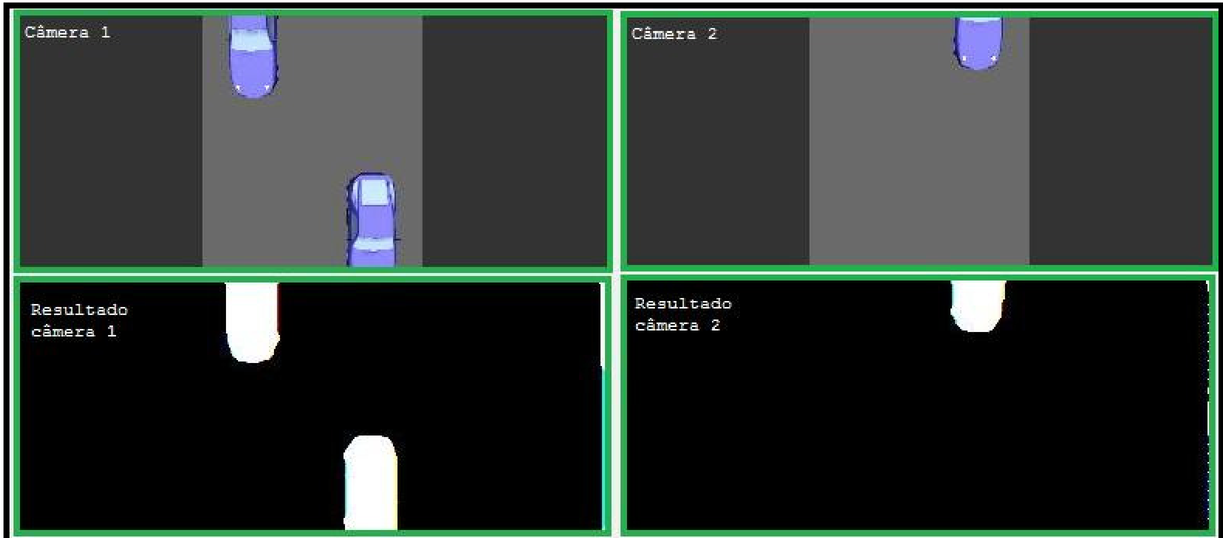


Figura 14 – Resultado aplicação algoritmo de detecção de movimento

O resultado do algoritmo de detecção de movimento é mostrado em uma imagem diferente para cada câmera representada pela aplicação. Conforme pode ser visto a aplicação do algoritmo traz como resultado um imagem binária onde as partes brancas estão em movimento e as partes pretas estão paradas.

Figura 15 – Resultado do tracking

O resultado da aplicação do algoritmo de *tracking* é apresentado com linhas que indicam o trajeto feito pelos objetos que estão sendo acompanhados. Cada veículo, quando entra na visão de uma câmera, tem o seu trajeto desenhado no quadro branco (linhas tracking), que está na parte inferior da interface. Pode-se verificar que os dois veículos apresentados na Figura 15 estão sendo acompanhados pelo sistema, conforme as duas linhas apresentadas.

3.4 RESULTADOS E DISCUSSÃO

Este trabalho apresenta a utilização de técnicas de visão computacional para a implementação de uma ferramenta de *tracking*, sendo esse feito entre várias câmeras. Como ponto inicial deste trabalho foi o usado o trabalho desenvolvido por Santos (2008). Deste trabalho foi utilizada a implementação do algoritmo NHD, e a partir dele foram implementadas as rotinas de *tracking*.

Para que fosse possível validar a implementação e verificar os resultados obtidos foi criado um ambiente de virtual em 3D que permite a simulação de veículos passando sobre uma estrada, e de onde pode-se adquirir imagens de câmeras virtuais.

Este capítulo foi separado em três seções, na secção 3.4.1 são feitas considerações referentes a opção de utilização de um ambiente virtual para a realização dos testes. Na secção 3.4.2 são apresentados resultados dos testes feitos no procedimento de *tracking*. Na secção 3.4.3 são feitas considerações referentes ao posicionamento das câmeras e sua interferência no acompanhamento dos objetos.

3.4.1 Ambiente de testes

Durante o desenvolvimento deste trabalho verificou-se que a aquisição de imagens para os testes seria um processo muito difícil, tendo em vista que seriam necessárias imagens de duas ou mais câmeras calibradas de uma estrada com movimento de carros satisfatório para os testes do sistema.

Na tentativa de solução para este problema teve-se a idéia de fazer os testes em um ambiente virtual. O grande problema do ambiente virtual é que não é possível validar a eficiência do algoritmo detecção de movimento em condições reais. Porém como o algoritmo NHD já havia sido testado em um ambiente real, inclusive com diversas condições de iluminação, e os algoritmos adicionados usam como entrada os resultados dele, foi feita a opção de não testar estas características.

Além disso, com o ambiente virtual é possível ter maior controle sobre as variáveis do ambiente, como número de veículos, velocidade, número de câmeras e posicionamento das mesmas. Manipular estas condições em um ambiente real seria uma tarefa que consumiria um esforço muito grande. Pelos motivos apresentados optou-se pela criação do ambiente de teste

virtual.

3.4.2 Testes do processo de *tracking*

Foram avaliadas fatores relacionados à execução do *tracking*. O primeiro foi o custo de execução do algoritmo de *tracking* em relação ao algoritmo NHD. O segundo foi o ganho obtido com o uso das técnicas do algoritmo de *OpticalFlow*. Também foi verificado o aumento do custo do *tracking* em relação à quantidade de objetos na tela.

Para a execução dos testes foram criadas quatro simulações, sendo a primeira com a apenas um veículo, a segunda e terceira com dois veículos (em posições diferentes) e a quarta com quatro veículos. Cada uma delas foi executada três vezes sendo uma sem execução do *tracking* para medir o custo da execução do algoritmo NHD, outras duas executando o *tracking*, uma com o recurso do *Optical flow* e outra sem.

Foi medido o tempo do processamento de cada frame de forma isolada para que a execução de outras atividades do sistema como a simulação do mundo interferissem na avaliação do desempenho. Depois de cada simulação foi feita a média do tempo de execução dos *frames*. Para a execução dos testes foi utilizado um computador com processador Pentium dual core T2370 de 1.73GHz e 2GB de memória RAM.

Depois dos dados terem sido coletados foram feitas algumas relações entre eles para que fosse possível visualizar o aumento de custo referente ao *tracking* e o ganho relacionado ao uso do *OpticalFlow*. O Quadro 5 mostra os dados coletados e suas relações, os dados que não são percentuais estão em milissegundos.

	Simulação 1	Simulação 2	Simulação 3	Simulação 4
NHD	125,7381	124,0348	122,9918	122,8319
Tracking	128,1031	127,8212	128,3361	128,7271
OpticalFlow	127,7897	126,7185	126,4816	126,5507
Tempo tracking	2,3650	3,7864	5,3443	5,8952
Tempo OptFlow	2,0516	2,6837	3,4898	3,7188
% aumento Tracking	1,8809%	3,0527%	4,3452%	4,7994%
% aumento OptFlow	1,6316%	2,1637%	2,8374%	3,0276%
% redução geral OptFlow	0,2446%	0,8627%	1,4450%	1,6907%
% redução tracking OptFlow	13,2516%	29,1226%	34,7005%	36,9182%

Quadro 5 – Resultados *tracking*

As colunas estão dispostas de forma que os casos de teste com maior número de pontos ao mesmo tempo no *tracking* estão mais a direita. Os casos dois e três têm o mesmo número de veículos, porém no segundo teste um veículo é identificado e o outro vem na sequência, ou seja, durante parte da simulação o *tracking* é feito com referência de um veículo e outra com dois. No outro caso de teste os carros estão em paralelo, o que significa que durante o *tracking* sempre haverão dois veículos na visão das câmeras.

Na primeira linha (NHD) são apresentados os tempos médios da execução do NHD apenas, na segunda linhas estão os tempos médios de execução do *tracking* sem o *OpticalFlow* (Tracking) e na terceira com (OpticalFlow).

A quarta linha mostra a diferença de tempo entre a execução do *tracking* e do NHD (Tempo tracking), e a quinta a diferença entre o *tracking* com o *OpticalFlow* e o NHD apenas (Tempo OptFlow). Nas sexta e sétima linhas (% aumento Tracking e % aumento OptFlow) esses valores são mostrados de forma percentual. Observando estes valores percebe-se que o algoritmo de *tracking* tem pouca influência no tempo de execução do sistema, e que para melhorar o desempenho geral do sistema o ideal seria otimizar o NHD. Também pode-se verificar que o tempo de execução do *tracking* aumentar conforme o número de pontos em movimento aumenta.

A oitava linha apresenta o percentual de ganho geral em termos do desempenho do uso da técnica de *OpticalFlow* (% redução geral OptFlow), e a nona linha apresenta o percentual de melhoria em relação apenas ao tempo do *tracking*. Observando-se esses dados é possível verificar que o uso do *OpticalFlow* melhora o tempo do sistema. O ganho em relação ao tempo geral é pequeno, porém o aumento do desempenho específica do *tracking* é razoável.

Pode-se verificar também que quanto maior o número de pontos, maior o ganho relacionado com o *OpticalFlow*, o uso desta técnica mostrou um melhora no desempenho do *tracking* por volta de 30%.

3.4.3 Posicionamento das câmeras

Para o teste dos algoritmos desenvolvidos foram colocadas no ambiente de teste duas câmeras com uma visão superior da estrada por onde os veículos passam. Para que a transferência de um objeto entre os *frames* funcione corretamente a posição das câmeras foram ajustadas manualmente, por tentativa e erro.

Durante esse processo foi observado que além do percentual de intersecção das imagens das câmeras a velocidade dos veículos também influencia na precisão do algoritmo. Percebeu-se que se aumentando a velocidade dos veículos a região de intersecção tem que ser aumentada, porém a relação entre esses dois valores que faça com que o algoritmo que faz o gerenciamento das câmeras funcione corretamente não foi descoberta.

Algumas análises ainda podem ser feitas em relação à questão das câmeras, além da relação entre intersecção das câmeras e velocidades dos veículos, outros fatores podem ser testas para verificar se coisas como tamanho dos objetos, por exemplo, interferem nessa relação. O número de câmeras poderia ser aumentado e elas poderiam ser dispostas em outras posições, não de forma linear e eqüidistante como foi utilizado nestes testes.

4 CONCLUSÕES

Este trabalho apresentou a implementação de um algoritmo de *tracking* e uma técnica que permite a melhoria do desempenho do mesmo. Também foi apresentada a implementação de um algoritmo que permite que o *tracking* seja efetuado através de várias câmeras.

O algoritmo de *tracking* foi implementado utilizando-se informações geradas pelo algoritmo NHD descrito em Santos (2008), que precisou ser modificado. Isto dificultou a implementação do *tracking*, já que foi necessário grande entendimento de sua implementação, mas trouxe a vantagem de que somente os pontos que estão em movimento fossem processados durante a execução do algoritmo.

A aplicação da técnica de *OpticalFlow* no algoritmo de *tracking* permitiu uma melhoria do desempenho na execução do sistema, principalmente se observamos apenas o custo de execução do *tracking*, onde a melhora ficou em torno de 30%. A implementação desta técnica se mostrou simples, demonstrando uma boa relação entre custo e benefício.

A implementação do *tracking* através de várias câmeras se mostrou funcional, inclusive para vários objetos na tela. Embora os testes tenham sido feitos usando apenas duas câmeras, o acréscimo de outras pode ser feito de maneira relativamente simples. Os testes efetuados foram feitos com posicionamento das câmeras de maneira comportada, as duas câmeras ficaram alinhadas em um ângulo próximo a 90º, sendo necessários mais estudos com câmeras em outras disposições.

Outro aspecto que não está relacionado diretamente com as técnicas apresentadas neste trabalho foi o ambiente de teste. Para que fosse possível efetuar os testes de várias câmeras sem grandes dificuldades na aquisição das imagens e o risco dessas imagens não serem conseguidas com a precisão desejada foi implementado um ambiente de testes virtual. O ambiente implementado se mostrou muito eficaz para os testes da aplicação permitindo, sem muito esforço, alterações nos objetos observados e também na disposição das câmeras que observam o ambiente.

4.1 EXTENSÕES

Este trabalho pode ser estendido de várias maneiras, tanto para melhorar as

funcionalidades existentes como para adicionar novas características. Algumas dessas funcionalidades são apresentas a seguir.

Neste trabalho o posicionamento das câmeras foi pouco explorado, pode-se testar o algoritmo buscando verificar até que ponto o que foi implementado funciona. Também devem ser buscadas técnicas que permitam maior liberdade para o posicionamento das câmeras. Podem ser estudadas técnicas para fazer a calibragem das câmeras em um ambiente real, para que seja possível a utilização de várias câmeras.

O processamento das várias visões do ambiente acontece em diversas *threads* buscando possibilitar o processamento paralelo. Um grande incremento para está característica seria a implementação da comunicação entre as *threads* via rede, permitindo, por exemplo, que cada câmera seja processada por um computador dedicado.

Pode-se adicionar ao sistema a capacidade de classificar os objetos identificados na cena em categorias de veículos, como por exemplo, carros, motos e caminhões. Também podem ser identificados alguns comportamentos não esperados na cena, como carros andando na contra mão ou em cima da calçada.

REFERÊNCIAS BIBLIOGRÁFICAS

BUTLER, Darren E. et al. **Real-time adaptive background segmentation.** Hindawi, 2005. Disponível em: <http://www.hindawi.com/getarticle.aspx?doi=10.1155/asp.2005.2292>. Acesso em: 29 mar. 2010.

CÉSAR JUNIOR, Roberto M.; COSTA, Luciano F. **Shape analysis and classification:** theory and practice. Boca Raton: CRC Press, 2001.

DENAMN, Simon; CHANDRAN, Vinod; SRIDHARAN, Sridha. **An adaptive optical flow technique for person tracking systems**. Brisbane, 2007. Disponível em: <http://eprints.qut.edu.au/14756/1/14756.pdf>. Acesso em: 12 nov. 2010.

DENMAN, Simon et al. **Multi-view intelligent vehicle surveillance system.** Brisbane, 2006. Disponível em: <http://portal.acm.org/citation.cfm?id=1190541>. Acesso em: 25 mar. 2010.

FACON, Jacques. **Processamento e análise de imagens.** Embalse: EBAI, 1993.

GARCIA, Mary L. **The design and evaluation of physical protection systems**. Burlington: Britsh Library, 2001.

GONZALEZ, Rafael C.; WOODS, Richard E. **Processamento de imagens digitais.** Tradução de Roberto Marcondes Cesar Junior, Luciano da Fontoura Costa. São Paulo: Edgard Blücher, 2008.

LOESCH, Cláudio; SARI, Solange T**. Redes neurais artificiais:** fundamentos e modelos. Blumenau: EdiFURB, 1996.

PARKER, Jim R. **Practical computer vision using C.** New York: John Wiley & Sons, 1994.

PORTAL MS. **Quantas câmeras gravam tudo o que você faz durante o dia?** [S.l.], 2009. Disponível em: <http://www.portalms.com.br/noticias/Quantas-cameras-gravam-tudo-o-que-voce-faz-durante-o-dia/Brasil/Tecnologia/959555674.html>. Acesso em: 19 mar. 2010.

QUIBT SYSTEMS. **Daniotrack:** video tracking software. [S.l.], [2010?]. Disponível em: <http://www.qubitsystems.com/Merchant2/merchant.mvc?Screen=PROD&Store_Code=QS&Product_Code=DV5>. Acesso em: 25 mar. 2010.

SANTOS, Daniel. **Sistema óptico para identificação de veículos em estradas**. 2008. 65 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SIMI REALITY MOTION SYSTEMS. **Simi MotionCapture3D:** 3D motion tracking system. [S.l.], 2010. Disponível em: <http://www.simi.com/en/products/mocap/index.html>. Acesso em: 25 mar. 2010.

TYAGI, Ambrish et al. **Fusion of multiple camera views for kernel-based 3D tracking**. [S.l.], 2007.Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.5917&rep=rep1&type=pdf>. Acesso em: 19 mar. 2010.