



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Auravitallis: Exploração de fluxo ótico na arte computacional

Leandro Ramalho Motta Ferreira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador

Prof. Dr. Marcelo Grandi Mandelli

Coorientador

Prof. Dr. Marcus Vinícius Lamar

Brasília
2017

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof. Dr. Marcelo Grandi Mandelli (Orientador) — CIC/UnB
Prof. Dr. Vinicius Ruela Pereira Borges — CIC/UnB
Prof. Dr.^a Suzete Venturelli — IdA/UnB

CIP — Catalogação Internacional na Publicação

Ramalho Motta Ferreira, Leandro.

Auravitallis: Exploração de fluxo ótico na arte computacional / Leandro Ramalho Motta Ferreira. Brasília : UnB, 2017.

112 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2017.

1. Visão Computacional, 2. Arte Computacional, 3. Fluxo Ótico,
4. Desempenho, 5. Raspberry PI, 6. Sistemas Integrados

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília-DF — Brasil

Dedicatória

Gostaria de dedicar em especial para minha sobrinha de 1 ano e 1 mês Lara Magno de Carvalho ,meu amigo Alexandre Dantas que infelizmente não está entre nós, essa sobreposição de eventos que ocorreram nesses 1 ano e 6 meses me ensina que a vida sempre continua, minha família que me apoiou em tudo e minha namorada Larissa Barbosa Nunes.

Agradecimentos

Agradeço aos meus pais, Fernando José e Lúcia Moreira, meu irmão Leonardo, minha irmã Liana por terem me dado todo apoio nesses anos de faculdade, minha namorada Larissa Barbosa Nunes por ter me dado todo amor e compreensão que podia pedir. Agradeço aos meus professores orientadores Marcelo Mandelli e Marcus Lamar por todo apoio, compreensão e orientação que deram. Agradeço Prahlada Hargreaves por ter trabalhado comigo na maior parte da fabricação da placa e da estrutura de madeira. Agradeço a todos os integrantes do laboratório MediaLab, Arthur Cabral e Guilherme Balduino. Agradeço em especial a doutora professora Suzete Venturelli por todo apoio e oportunidades que me foi dado. Agradeço a todos meus amigos da CJR - empresa júnior de computação.

Resumo

Esse trabalho consiste da criação e implementação de uma nova versão do projeto Auravittallis. Utiliza-se placas de circuito impresso, plataformas de desenvolvimento e a linguagem de programação *Processing*. A obra consiste de um ambiente de vida celular computacional representado por uma matriz 5x10 de *LEDS*, a obra mede aproximadamente dois metros de altura por um metro de largura. A obra possui uma câmera que capta a imagem. O movimento do interator gera alimentos no ambiente celular, esse movimento é estimado pelos algoritmo de fluxo ótico. A vida celular computacional é gerenciado por um software embarcado em uma plataforma de desenvolvimento *Raspberry Pi*. O *software* foi implementado em três versões utilizando diferentes algoritmos de fluxo ótico. Estuda-se as implementações para a interação da obra, métodos *Horn & Schunck* e expansão polinomial. Compara-se três implementações *Horn & Schunck* não paralelizada, expansão polinomial usando *Opencv* e *Horn & Schunck* paralelizada. Verifica-se que dependendo dos casos a implementações diferentes são adequadas. A obra também foi apresentada em encontros de arte e tecnologia.

Palavras-chave: Visão Computacional, Arte Computacional, Fluxo Ótico, Desempenho, Raspberry PI, Sistemas Integrados

Abstract

This work consists of the creation and implementation of a new version of the Auravitallis project. It uses printed circuit boards, development platforms and the programming language *Processing*. The piece consists of a computational cellular life environment represented by a 5x10 matrix of LEDs, the piece measures approximately two meters high by one meter wide. The work has a camera that captures the image. The interactor drive generates food in the cellular environment, this movement is estimated by the optical flow algorithms. Computational cellular life is managed by software embedded in a *Raspberry Pi* development platform. Software has been implemented in three versions using different optical flow algorithms. It is studied the implementations for the interaction of the work, methods *Horn & Schunck* and polynomial expansion. The three implementations Horn & Schunck not parallelized, polynomial expansion using OpenCV and Horn & Schunck parallelized are compared. It turns out that depending on the cases the different implementations are appropriate. The piece was also presented at art and technology meeting.

Keywords: Computer Art, Computer Vision, Optical Flow, Performance, Raspberry PI, Integrated Systems

Sumário

1	Introdução	1
1.1	Objetivo	2
1.1.1	Objetivos Gerais	2
1.1.2	Objetivos Específicos	3
1.2	Justificativa	3
1.3	Metodologia da Pesquisa	3
1.4	Estrutura do texto	3
2	Fundamentação Teórica	5
2.1	Visão Computacional e <i>Optical Flow</i> - Fluxo Ótico	5
2.1.1	Visão Computacional Conceito e Aplicações	5
2.1.2	Fluxo Ótico	6
2.2	Métodos de estimativa de fluxo ótico	7
2.2.1	Nomenclatura Básica	7
2.2.2	Suposição da Constância de Brilho	8
2.2.3	Método Clássico Horn & Schunck	9
2.2.4	Expansão Polinomial	12
2.2.5	Endpoint Error	14
2.2.6	Performance de algoritmos de estimativa de fluxo ótico	15
2.3	Plataformas de Desenvolvimento	16
2.3.1	Raspberry Pi	16
2.3.2	Comparação Entre <i>Arduino</i> e Raspberry PI	17
2.3.3	Comparação entre Raspberry e BeagleBone	18
2.4	Processing	18
2.4.1	Definição e Aplicações	18
2.4.2	Biblioteca Hardware	19
2.5	OpenCV	21
2.6	Placas de Circuito Impresso (PCB) e circuitos integrados	24
2.6.1	Definição	24

2.6.2	Camada A	24
2.6.3	Camada de metal condutor	24
2.6.4	<i>Shift Register 745HC595</i>	25
2.7	Trabalhos Relacionados	26
3	Materiais e Métodos	28
3.1	<i>Hardware</i>	29
3.1.1	Raspberry Pi	30
3.1.2	PCB	31
3.2	Software	38
3.2.1	Comportamento Celular	40
3.2.2	Captura de Imagem	46
3.2.3	Interface de Hardware	48
3.2.4	Implementações de estimativa de fluxo ótico	49
3.3	Manufatura Carcaça	54
4	Resultados e Análise	55
4.1	Resultados	55
4.1.1	Tempo de processamento	55
4.1.2	Verificação e validação dos algoritmos de fluxo ótico	70
4.1.3	Interatividade da Obra	71
4.2	Apresentação da Obra	72
5	Conclusões	74
5.1	Conclusões	74
5.2	Trabalhos Futuros	75
	Referências	76
	Anexo	78
I	Anexo Geral	79
II	Anexo Dataset KITTI	82
III	Anexo Dataset Middlebury	87
IV	Anexo Interação	94
V	Anexo Apresentacao Museu	97

Lista de Figuras

2.1	Foto aérea exemplo problema de visão computacional [1]	6
2.2	Exemplo de aplicação em robótica móvel [2]	7
2.3	Problema de Fluxo Óptico e Problema de <i>Aperture</i> [3]	9
2.4	Exemplo Opencv	21
2.5	Esquema Conceitual de uma Placa de Circuito Impresso - PCB. 1 indica a camada A e 2 indica a camada B	24
2.6	Shift register 745HC595	25
3.1	Foto da obra sendo testada.	29
3.2	Raspberry Modelo 2B	30
3.3	Esquemático lógico simplificado da PCB, original em anexo I.1	32
3.4	Esquema da PCB	33
3.5	Imagem do Projeto invertida horizontalmente preenchida e negativada	34
3.6	Prensando o papel couchê na placa de cobre	34
3.7	Banho de perclorato de ferro	35
3.9	Resultado do processo do banho de perclorato de ferro.	36
3.8	Utilização da micro retífica na placa	36
3.10	Soldagem dos componentes	37
3.11	Camada Inferior após correções	37
3.12	Camada Superior após correções	38
3.13	Software Horn & Schunck Serial.	39
3.14	Software expansão polinomial/OpenCV.	39
3.15	Software Horn & Schunck paralelo.	40
3.16	Exemplo de movimento celular para o vizinho Nordeste	41
3.17	Exemplo de farejar alimento	42
3.18	Estado T_0 , antes de um surgimento de uma nova alimento e o estado T_1 , depois do surgimento de uma nova alimento	45
4.1	Frames seguidos de exemplos do dataset [4]	56

4.2	Gráfico de número do teste por tempo de processamento <i>dataset Middlebury</i> , método expansão polinomial.	57
4.3	Gráfico de número do teste por tempo de processamento <i>dataset Middlebury</i> , método Horn & Schunck.	57
4.4	Imagem número 00 frame 11	62
4.5	Gráfico de número do teste por tempo de processamento <i>dataset KITTI</i> , método Horn & Schunck.	63
4.6	Gráfico de número do teste por tempo de processamento <i>dataset KITTI</i> , método expansão polinomial.	63
4.7	Imagem <i>dataset</i> interação.	68
4.8	Corretude utilizando <i>dataset</i> próprio. A: O fluxo calculado pintado com cores correspondentes as direções na imagem. B: Segundo frame. C: Primeiro frame. As cores indicam a direção de acordo com o circulo na imagem	71
4.9	Fazendo a instalação da obra e iluminação.	72
4.10	A obra montada de frente.	73
I.1	Esquemático lógico da placa	80
I.2	Tabela relação pinagem <i>RaspberryPI 2B</i>	81
II.1	Imagem 0 1242 x 375	82
II.2	Imagem 1 1242 x 375	82
II.3	Imagem 2 1242 x 375	82
II.4	Imagem 3 1242 x 375	83
II.5	Imagem 4 1242 x 375	83
II.6	Imagem 5 1242 x 375	83
II.7	Imagem 6 1242 x 375	83
II.8	Imagem 7 1242 x 375	83
II.9	Imagem 8 1242 x 375	84
II.10	Imagem 9 1242 x 375	84
II.11	Imagem 10 1242 x 375	84
II.12	Imagem 11 1242 x 375	84
II.13	Imagem 12 1242 x 375	84
II.14	Imagem 13 1242 x 375	85
II.15	Imagem 14 1242 x 375	85
II.16	Imagem 15 1242 x 375	85
II.17	Imagem 16 1242 x 375	85
II.18	Imagem 17 1242 x 375	85
II.19	Imagem 18 1242 x 375	86

II.20 Imagem 19 1242 x 375	86
III.1 Army 584 x 388	87
III.2 Bakyard 584x388	88
III.3 Basketball 640x480	88
III.4 Dumptruck 640x480	89
III.5 Evergreen 584x388	89
III.6 Grove 640x480	90
III.7 Mequon 640x480	90
III.8 Schefflera 640x480	91
III.9 Teddy 420x380	91
III.10Urban 640x480	92
III.11Wooden 640x480	92
III.12Yosemite 640x480	93
IV.1 Imagem Interação Nordeste 160 x 120	94
IV.2 Imagem Interação Noroeste 160 x 120	95
IV.3 Imagem Interação Sudeste 160 x 120	95
IV.4 Imagem Interação Sudoeste 160 x 120	96

Lista de Tabelas

2.1	Erros da implementação Horn-Schunck [5]	15
2.2	Erros da implementação expansão polinomial [5]	16
2.3	Tabela de Modelos Raspberry Pi Disponíveis [6]	17
2.4	Pinagem do Shift register 74HC595	25
2.5	Lista de trabalhos utilizando visão computacional e placas de desenvolvimento.	26
2.6	Lista de trabalhos e quais algoritmos foram usados.	27
3.1	Lista de peças para Fabricação da placa circuito Impresso	31
4.1	Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn Schunck, <i>dataset Middlebury</i> de Avaliação, Quad-Core ARM cortex-A7 900Mhz e 1 GB RAM.	58
4.2	Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn & Schunck, <i>dataset Middlebury</i> , processador core i7 -3635 @ 2,4 GHz e RAM 8 GB	59
4.3	Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn & Schunck utilizando <i>Threads, dataset Middlebury</i> , Quad-Core ARM cortex-A7 900Mhz e 1 GB RAM	60
4.4	Tempo de processamento médio em milisegundos pelos métodos Horn & Schunck Serial e Horn & Schunck Paralelo utilizando <i>Threads, dataset Middlebury</i> , Quad-Core ARM cortex-A7 ARM cortex-A7 900MHz e 1 GB RAM	61
4.5	Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn & Schunck, <i>dataset KITTI</i> .	64
4.6	Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn & Schunck, <i>dataset KITTI</i> , Hardware core i7 -3635 @ 2,4 GHz	65
4.7	Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn & Schunck paralelizado, <i>dataset KITTI</i> .	66
4.8	Tempo de processamento médio em milisegundos pelos métodos <i>Horn & Schunck</i> paralelizado e <i>Horn & Schunck</i> não paralelizado (serial), <i>dataset KITTI</i> .	67

4.9	Benchmark dataset próprio. Processado Raspberry Modelo 2B Quad-Core ARM cortex-A7 @ 900MHz	69
4.10	Benchmark dataset próprio, <i>Horn & Schunck</i> paralelizado. Processado Raspberry Modelo 2B Quad-Core ARM cortex-A7 @ 900MHz	69
4.11	Benchmark dataset próprio, <i>Horn & Schunck</i> paralelizado e <i>Horn & Schunck</i> não paralelizado (serial). Processado Raspberry Modelo 2B Quad-Core ARM cortex-A7 @ 900MHz	70

Capítulo 1

Introdução

Visão Computacional é uma área que possui como objetivo a formulação científica de modelos computacionais do sistema visual humano [7]. A área de visão computacional possui também como objetivo criação de sistemas autônomos que podem desempenhar tarefas as quais o sistema visual humano consegue resolver. Exemplos de problemas que são resolvidos são, a detecção de navios militares em uma foto, a detecção de movimento em um *streaming* de vídeo, cálculo de deslocamento de um robô através de uma câmera instalada nele [7].

Dentre um dos problemas é o problema de cálculo de movimento de todos os objetos em uma sequência de imagens. Esse problema é chamado de estimativa de fluxo ótico. Exemplo de aplicações para a resolução de fluxo ótico são *odometria* na robótica, detecção de movimento em um *streaming*. Outra aplicação de fluxo ótico é arte pós moderna [8]. O fluxo ótico pode ser utilizado como uma forma natural de interação com uma obra de arte interativa. Por exemplo, uma técnica de estimativa de fluxo ótico pode ser utilizada para a detecção do movimento de uma pessoa interagindo com uma obra.

Este trabalho utiliza um projeto de arte como prova do conceito do uso de fluxo ótico em aplicações de arte pós-moderna. A arte a partir do movimento pós modernista começa a expandir instrumentos para expressar poéticas. A ciência da computação também pode ser uma dessas ferramentas para expressar ideias. Outro fator vantajoso é que a arte também tem é uma maneira lúdica de apresentar conceitos de tecnologia para pessoas. Portanto, por esses fatores utilizamos a arte como prova de conceito para explorar conceitos de ciência da computação. Os conceitos a serem explorados nessa prova de conceito são visão computacional, sistemas embarcados e placas de circuito impresso.

Iniciado como um projeto de iniciação científica no laboratório de pesquisa de arte computacional Medialab UnB, o trabalho de iniciação científica se tornou esse trabalho de graduação. O projeto de iniciação científica foi estendido como uma parceria entre o laboratório Medialab da Universidade de Brasília e o Departamento de Ciência de Computação(CIC). A poética e ideia inicial foi concebida no ambiente de pesquisa do Medialab.

O projeto inicial, chamado de *Auravitallis*, feito no projeto de iniciação científica consiste de um ambiente de vida celular computacional representado por um totem de madeira de dois (2) metros de altura e um (1) metro de largura. O projeto inicial contém uma matriz 5x10 de *LEDS*. Cada *LED* aceso representa uma célula viva no ambiente celular. O totem possui uma câmera que capta a existência de movimento, caso haja movimento há reprodução celular. A reprodução na obra é a criação de outras células na memória do programa portanto surgem mais *LEDS* acesos. A vida celular computacional é gerenciado por um software embarcado na plataforma de desenvolvimento *Raspberry Pi*. A plataforma de desenvolvimento faz interface com os *LEDS* através dos pinos de entrada/saída de propósito geral, em inglês *General Purpose Input/Output* (GPIO). Os sinais transmitidos pelos pinos GPIO são convertidos em uma placa de circuito impresso, onde é ligada a matriz de *LEDS* [8].

Este projeto é uma extensão do projeto inicial *Auravitallis*, onde a detecção de movimento simples é alterada para estimativa de fluxo ótico. A nova interação gera alimentos no ambiente onde o fluxo ótico é grande o suficiente. A existência de alimento no ambiente celular atrai as células e elas se deslocam em sua direção. A vida celular computacional é gerenciado por um software embarcado em uma plataforma de desenvolvimento *Raspberry Pi*. O *software* foi implementado em três versões utilizando diferentes algoritmos de fluxo ótico. Estuda-se as implementações para a interação da obra, métodos *Horn & Schunck* e expansão polinomial. Compara-se três implementações *Horn & Schunck* não paralelizada, expansão polinomial usando *Opencv* e *Horn & Schunck* paralelizada.

1.1 Objetivo

Nesta Seção serão apresentados os objetivos gerais e específicos do projeto. Esses dois objetivos são descritos nas Subseções 1.1.1 e 1.1.2 respectivamente.

1.1.1 Objetivos Gerais

O objetivo geral desse trabalho consiste da criação da obra de arte *Auravitallis* como uma prova de conceito arte computacional. A obra de arte utiliza algoritmos de visão computacional, microcontroladores, fitas de LEDs, placas de circuito impresso. A obra tem objetivo aproximar o público dos conceitos de visão computacional. De forma concreta o trabalho será uma estrutura de madeira a qual contém circuitos, uma matriz de fitas de LEDs 5 x 10, microcontroladores e uma câmera que estará na parte frontal da estrutura da madeira visando o público. O microcontrolador estará programado para calcular o fluxo ótico e gerar uma interação com o público. Após a explicação breve dos objetivos gerais se inicia a escrita dos objetivo específicos.

1.1.2 Objetivos Específicos

Como objetivos específicos podem ser citados:

- Programação de um protótipo no microcontrolador que capta o fluxo ótico;
- Projetar o circuito *LEDs*, microcontroladores e circuitos integrados;
- Configuração do microcontrolador para necessitar o mínimo de periféricos para controlá-lo;
- Fabricação, solda e instalação do circuito na estrutura de madeira;
- Criação do primeiro protótipo da obra completo com circuito, análise e estudo da interação;
- Refinamento da obra e interação;
- Implementação e investigação de métodos para estimativa de fluxo ótico.

1.2 Justificativa

Este trabalho promove a integração do departamento de ciência da computação com o departamento de artes, sendo ambos da Universidade de Brasília (UnB). Portanto utiliza-se a ciência da computação para cumprir com o objetivo e propósito da arte. Outro aspecto é que como a arte é inclusiva, inclui o público na ciência da computação e os algoritmos estudados nesse trabalho. A arte também é uma maneira lúdica de apresentar conceitos de tecnologia. Portanto esses fatores utilizamos a arte como prova de conceito para explorar algoritmos de ciência da computação.

1.3 Metodologia da Pesquisa

Inicia-se o trabalho de graduação explorando o artigo do projeto Auravittalis original [8]. Após esse passo inicial, estudam-se os métodos de fluxo ótico, implementações para a ferramenta já utilizada no projeto original e também é estudada a tecnologia das placas de circuito impresso. Uma vez implementada uma nova versão do *software* e, fabricada a placa de circuito impresso, são iniciados testes para avaliar a melhor forma de interação. Após decidida a interação são feitos testes e tempo de processamento. Terminadas todas essas etapas cria-se o texto em questão.

1.4 Estrutura do texto

O texto que descreve este trabalho graduação é dividido em cinco capítulos.

- **Introdução**
O Capítulo 1, introduz o método científico utilizado e descreve estrutura do texto em geral.
- **Fundamentação Teórica**
O Capítulo 2 fundamenta teoricamente os conceitos utilizados e explora ferramentas e suas características.
- **Materiais e Métodos**
O Capítulo 3 analisa o software construído, configurações do *Raspberry Pi*, fabricação da *PCB*, estrutura de madeira e o processo de apresentação da obra.
- **Resultados e Análise**
O Capítulo 4 apresenta as análises desempenho do software de fluxo óptico implementado, resultado da experiência da interação e a apresentação da obra.
- **Conclusões**
O Capítulo 5 traz as conclusões do trabalho e sugere idéias para novos trabalhos.

Capítulo 2

Fundamentação Teórica

Esse capítulo apresenta conceitos e ferramentas que serão utilizados no trabalho. Dessa forma, o capítulo é dividido como segue. A Seção 2.1 consiste na exploração e explanação de conceitos relativos à visão computacional. A Seção 2.2 aborda dois métodos de estimativa de fluxo ótico e mensuração de erros desses algoritmos. A Seção 2.3 apresenta conteúdo sobre a plataforma de desenvolvimento *Raspberry Pi* e comparações com outras opções. A Seção 2.4 apresenta o ambiente de programação e linguagem de programação utilizada no trabalho. A Seção 2.5 aborda a biblioteca aberta de visão computacional *OpenCV*. A Seção 2.6 apresenta a conceituação de placas de circuito impresso, PCB ou PCI. Por fim, a Seção 2.7 demonstra trabalhos semelhantes que são utilizados como inspiração e referência.

2.1 Visão Computacional e *Optical Flow* - Fluxo Ótico

Nesta Seção é explorado o conceito de visão computacional e um problema em específico, o cálculo do fluxo ótico. A Subseção 2.1.1 descreve o conceito de visão computacional e ilustra esse conceito através de aplicações. A Subseção 2.1.2 explora o conceito de fluxo ótico, seu significado e exemplos de aplicações.

2.1.1 Visão Computacional Conceito e Aplicações

Visão Computacional é uma área que possui como objetivo a formulação científica de modelos computacionais do sistema visual humano [7]. A área de visão computacional possui também como objetivo a criação de sistemas autônomos que possam desempenhar tarefas similares ao sistema visual humano [7]. Por exemplo, um ser humano recebe a missão de encontrar navios em uma foto aérea conforme mostrado na Figura 2.1.

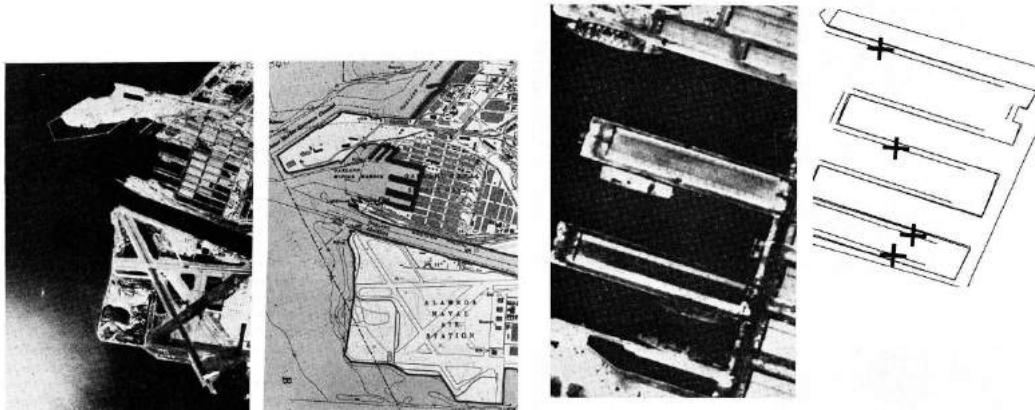


Figura 2.1: Foto aérea exemplo problema de visão computacional [1]

O sistema visual humano consegue decidir como varrer a foto, como evitar a falha branca na imagem e/ou se percorre a costa apresentada. Portanto, categoriza-se esse problema como de visão computacional. Dito isso nessa pesquisa, concentra-se é determinar deslocamento que ocorre entre dois momentos em um vídeo, o que se considera um problema de fluxo ótico.

2.1.2 Fluxo Ótico

Fluxo ótico consiste da detecção e modelagem do deslocamento de elementos em uma sequência de imagens. Esse deslocamento é considerado no plano de duas dimensões de uma sequência de imagens. Exemplifica-se aplicações de fluxo ótico como a estimativa de deslocamento de objetos/pessoas/elementos em uma imagem, essa estimativa é baseada em duas ou mais imagens. Fluxo ótico é uma tarefa apresentada em contextos como robótica [9]. A maior parte dos algoritmos são baseados em 2 frames, a partir da primeira iteração dos algoritmos ele se repete para todos os outros instantes seguintes[9].

Para uma melhor ilustração do problema que o fluxo ótico representa exemplificam-se os seus usos. O primeiro está relacionado à área de robótica, rastreamento de objetos, detecção de movimento, navegação do robô e odometria visual [10].

Um exemplo mais detalhado é o trabalho feito por Jörg Rett e Jorge Dias, de navegação do robô chamado *Nicole 2.2*. O trabalho consiste de uma integração entre o método de estimativa de fluxo ótico e *FOE* (Foco de Expansão de uma imagem) 2.2. Cria-se um ponto de referência, o *FOE*, e captura-se o movimento através de uma câmera e a estimativa do fluxo ótico para se obter o quanto o robô se movimentou na direção daquele ponto. O cálculo é obtido através de uma projeção na direção do *FOE*. A Figura 2.2 ilustra o problema a ser resolvido pelo fluxo ótico [2]. Após uma introdução sobre aplicações de conceitos da visão computacional, abordam-se os métodos de estimativa de fluxo ótico.

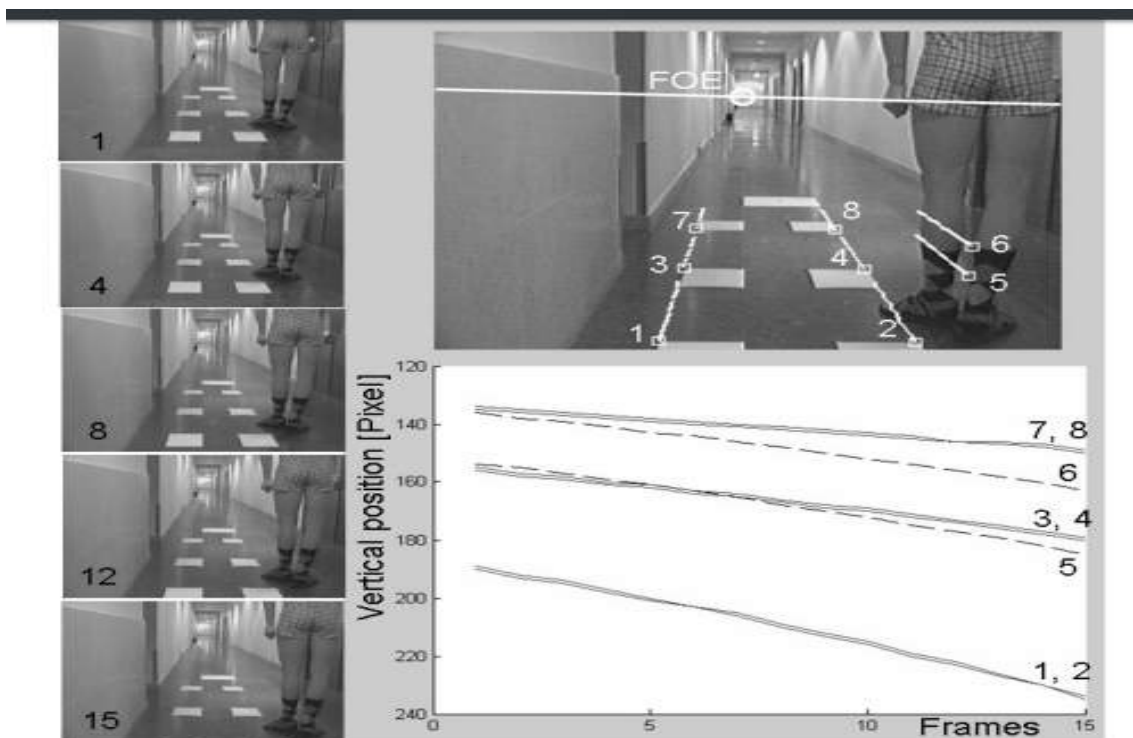


Figura 2.2: Exemplo de aplicação em robótica móvel [2]

2.2 Métodos de estimativa de fluxo ótico

Essa Seção aborda alguns métodos de estimativa do fluxo ótico. Detalha-se melhor dois desses métodos que foram utilizados nesse trabalho, o método *Horn-Schunck* e *Expansão Polinomial*. Esta Seção é subdividida como se segue. A Subseção 2.2.1 define nomenclaturas que são necessárias para compreensão dos algoritmos. A Subseção 2.2.2 aborda uma suposição chave do método *Horn-Schunck*. A Subseção 2.2.3 segue a ideia da suposição da constância de brilho e o problema de abertura e formula um algoritmo. A Subseção 2.2.4 apresenta o conceito básico de expansão polinomial que é base do segundo algoritmo. A Subseção 2.2.5 explica uma mensuração para correteude dos algoritmos de fluxo ótico. A Subseção 2.2.6 explora estudos de performance entre os dois algoritmos utilizados neste trabalho.

2.2.1 Nomenclatura Básica

Define-se um vídeo como uma sequência de imagens. Essas imagens que formam o vídeo são chamadas de quadros ou *frames*, por sua vez são compostos por *pixels*. Os pixels estão mapeados em formato da função de intensidade $I(x,y,t)$, sendo $x,y,t \in \mathbb{N}$. Cada valor $I(x,y,t)$ é um valor de brilho naquele pixel, que pode ser entendido como sua cor. Definimos uma trajetória de um ponto p como $(x(t),y(t))$. Sendo x,y par de coordenadas com ponto de origem

na imagem, sendo t o tempo em segundos. Essa introdução de nomenclatura é necessária para o próximo conceito a ser utilizado para a estimativa do fluxo ótico, a suposição da constância de brilho.

2.2.2 Suposição da Constância de Brilho

A suposição da constância de brilho ou constância de cor é assumir que dado qualquer pixel na imagem ele não sofrerá mudança de brilho independentemente de movimentações [3][9]. Matematicamente define-se da seguinte forma:

$$I(x(t), y(t), t) \rightarrow \frac{dI}{dt} = 0 \quad (2.1)$$

A Equação 2.1 não considera pixels que não estão na imagem no ponto no instante $t + \partial t$. Sendo \vec{h} o vetor de deslocamento no referencial da imagem. Expressa-se a suposição da constância do brilho de outra forma:

$$\vec{h}(x, y) = (u, v) = \left(\frac{dx}{dt}, \frac{dy}{dt} \right) \quad (2.2)$$

$$\nabla I = I_x + I_y + I_t = \frac{dI}{dx} \cdot \frac{dx}{dt} + \frac{dI}{dy} \cdot \frac{dy}{dt} + I_t \Rightarrow \nabla I \cdot h + I_t = 0 \quad (2.3)$$

$$I_x \cdot \frac{dx}{dt} + I_y \cdot \frac{dy}{dt} + I_t = I_x \cdot u + I_y \cdot v + I_t = 0 \quad (2.4)$$

$$-\frac{I_x \cdot u + I_t}{I_y} = v \quad (2.5)$$

A partir da Equação 2.5 é possível desenhar em um gráfico de uma função de primeira ordem. O vetor \vec{h} é composto de um par ordenado de deslocamento horizontal u e vertical v . Sendo o ∇I o gradiente da função $I(x, y, t)$. Sendo u, v respectivamente velocidade horizontal e velocidade vertical no referencial da imagem. A solução está definida em algum ponto da reta, mas não possuímos equações linearmente independentes suficiente para determinar qual desses pontos da reta seria o fluxo u e v , portanto é necessário uma estimativa para determinar o fluxo de cada pixel em duas imagens separadas por um instante dt . O gráfico que ilustra a Equação 2.3 é mostrado na Figura 2.3.

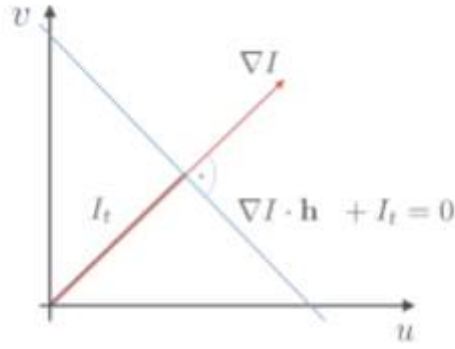


Figura 2.3: Problema de Fluxo Óptico e Problema de *Aperture* [3]

Dada as conclusões obtidas pela suposição de constância de brilho podemos introduzir um primeiro método que utiliza essa estimativa, o método Horn-Schunck.

2.2.3 Método Clássico Horn & Schunck

O método Horn & Schunck é empregado para estimar o deslocamento dos elementos de uma sequência de imagens. A variação das imagens pode ser definida como o vetor estimado $\vec{h} = (u, v)$. As variáveis u, v são coordenadas horizontais e verticais do fluxo na imagem. O método de estimar pode ser resumido em encontrar o vetor \vec{h} que minimiza a função $J(h)$. Define-se a função $J(h)$ da seguinte forma 2.6.

$$J(h) = \int_{\Omega} (I_x \cdot u + I_y \cdot v + I_t)^2 + \alpha^2 (|\nabla u|^2 + |\nabla v|^2) \quad (2.6)$$

Sendo I_x, I_y, I_t a derivada de primeira ordem parcial na coordenada x, y, t respectivamente. Sendo também $\nabla u, \nabla v$ o gradiente das coordenadas horizontais e verticais. Também é definido α um fator de suavidade do algoritmo, isto é quanto maior mais suave o fluxo óptico estimado [3].

Onde α é um parâmetro de controle da suavidade. A minimização da função pode ser expressada com as seguintes equações *Euler-Lagrange*:

$$I_x^2 u + I_x I_y v = \alpha^2 \text{div}(\nabla u) - I_x I_t, \quad (2.7)$$

$$I_x u + I_x I_y^2 v = \alpha^2 \text{div}(\nabla v) - I_x I_t \quad (2.8)$$

Sendo o $\text{div}(\nabla)$ o Laplaciano, que pode ser aproximado da seguinte forma:

$$\text{div}(\nabla u) \simeq \bar{u} - u \quad (2.9)$$

$$\text{div}(\nabla v) \simeq \bar{v} - v \quad (2.10)$$

Onde os vetores (\bar{u}, \bar{v}) são a média local. No caso discreto o Laplaciano é facilmente calculado. Para isso, substituímos a aproximação do Laplaciano nas Equações 2.7 e 2.8, como visto nas Equações 2.11 e 2.12.

$$(\alpha^2 + I_x^2 + I_y^2)(\bar{u} - u) = -I_x(I_x\bar{u} + I_y\bar{v} + I_t) \quad (2.11)$$

$$(\alpha^2 + I_x^2 + I_y^2)(\bar{v} - v) = -I_y(I_x\bar{u} + I_y\bar{v} + I_t) \quad (2.12)$$

É necessário calcular I_x^2 , I_y^2 e I_t^2 , pixel a pixel para estimativa dos vetores $h = (\bar{u}, \bar{v})$. O caso numérico que é explorado nesse trabalho será abordado no próxima Subseção.

2.2.3.1 Discretização Horn-Schunck

Numericamente o cálculo das derivadas parciais é realizado através de operações de convolução com matrizes dos dois frames e médias locais, sendo $p_{m,n}$ o píxel na linha m e coluna n . Essa matrizes são chamadas de matrizes kernel, dadas por.

$$P = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{pmatrix}, \quad (2.13)$$

$$Q = \begin{pmatrix} q_{1,1} & \cdots & q_{1,n} \\ \vdots & \ddots & \vdots \\ q_{m,1} & \cdots & q_{m,n} \end{pmatrix} \quad (2.14)$$

P sendo frame de pixels do momento anterior $t - 1$ e Q frame de pixels do momento atual t . Deste modo obtém-se:

$$I_x = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{pmatrix} * \begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} \cdot \frac{1}{4} + \begin{pmatrix} q_{1,1} & \cdots & q_{1,n} \\ \vdots & \ddots & \vdots \\ q_{m,1} & \cdots & q_{m,n} \end{pmatrix} * \begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} \cdot \frac{1}{4} \quad (2.15)$$

$$I_y = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} \cdot \frac{1}{4} + \begin{pmatrix} q_{1,1} & \cdots & q_{1,n} \\ \vdots & \ddots & \vdots \\ q_{m,1} & \cdots & q_{m,n} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} \cdot \frac{1}{4} \quad (2.16)$$

$$I_t = \begin{pmatrix} p_{1,1} & \cdots & I_{1,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & I_{m,n} \end{pmatrix} * \begin{pmatrix} -1 & -1 \\ -1 & -1 \end{pmatrix} \cdot \frac{1}{4} + \begin{pmatrix} q_{1,1} & \cdots & q_{1,n} \\ \vdots & \ddots & \vdots \\ q_{m,1} & \cdots & q_{m,n} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \cdot \frac{1}{4} \quad (2.17)$$

As médias locais de vetores (\bar{u}, \bar{v}) são calculadas de uma forma semelhante, mas apenas com o resultado do frame atual Q , pois são os vetores u e v do momento atual que estão sendo calculados.

$$\bar{u} = \begin{pmatrix} \bar{u}_{1,1} & \cdots & \bar{u}_{1,n} \\ \vdots & \ddots & \vdots \\ \bar{u}_{m,1} & \cdots & \bar{u}_{m,n} \end{pmatrix} = \begin{pmatrix} u_{1,1} & \cdots & u_{1,n} \\ \vdots & \ddots & \vdots \\ u_{m,1} & \cdots & u_{m,n} \end{pmatrix} * \begin{pmatrix} \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \\ \frac{1}{6} & 0 & \frac{1}{6} \\ \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \end{pmatrix} \quad (2.18)$$

$$\bar{v} = \begin{pmatrix} \bar{v}_{1,1} & \cdots & \bar{v}_{1,n} \\ \vdots & \ddots & \vdots \\ \bar{v}_{m,1} & \cdots & \bar{v}_{m,n} \end{pmatrix} = \begin{pmatrix} v_{1,1} & \cdots & v_{1,n} \\ \vdots & \ddots & \vdots \\ v_{m,1} & \cdots & v_{m,n} \end{pmatrix} * \begin{pmatrix} \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \\ \frac{1}{6} & 0 & \frac{1}{6} \\ \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \end{pmatrix} \quad (2.19)$$

A solução apresentada para o sistemas definido pelas equações 2.11 e 2.12 é encontrada iterativamente dependendo de um cálculo anterior e a condição de parada desse método iterativo é descrito como uma soma da diferença entre valores consecutivos entre iterações [3]:

$$u^{n+1} = \bar{u}^n - I_x \frac{I_x \bar{u}^n + I_y \bar{v}^n + I_t}{\alpha^2 + I_x^2 + I_y^2} \quad (2.20)$$

$$v^{n+1} = \bar{v}^n - I_x \frac{I_x \bar{u}^n + I_y \bar{v}^n + I_t}{\alpha^2 + I_x^2 + I_y^2} \quad (2.21)$$

$$\frac{1}{N} \sum_{i,j} (u_{i,j}^{n+1} - u_{i,j}^n)^2 + (v_{i,j}^{n+1} - v_{i,j}^n)^2 < \epsilon^2 \quad (2.22)$$

As Equações 2.20 e 2.21 indicam o método iterativo onde o próximo valor do fluxo no pixel $\vec{h}^{n+1} = (\bar{u}^{n+1}, \bar{v}^{n+1})$ depende do valor calculados anteriormente $\vec{h}^n = (\bar{u}^n, \bar{v}^n)$. A Equação 2.21 representa a condição de parada, se a afirmação for verdadeira o método iterativo é finalizado, o critério é baseado na diferença ser decrescente a cada iteração, sendo ϵ^2 o valor de parada [3]. Terminado o método clássico de *Horn-Schunck* explora-se a otimização utilizando convolução separável.

2.2.3.2 Convolução Separável

Uma das operações necessárias para a estimativa de fluxo ótico é a convolução. Um caso da operação de convolução é a convolução separável. As convoluções do tipo separável são comu-

tativas. [11]. Explora-se a prova da convolução separável nas equações 2.23, 2.24, 2.25 e 2.26. Dada a definição de convolução duas dimensões [11]:

$$y[m, n] = x[m, n] * h[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} h[i, j] x[m-i, n-j] \quad (2.23)$$

Sendo a matriz separável $h[m, n] = h_1[m]h_2[n]$ substitui-se na equação 2.24 abaixo:

$$y[m, n] = x[m, n] * h[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[m-i, n-j] h_1[i] h_2[j] \quad (2.24)$$

$$y[m, n] = x[m, n] * h[m, n] = \sum_{j=-\infty}^{\infty} h_2[j] \sum_{i=-\infty}^{\infty} x[m-i, n-j] h_1[i] \quad (2.25)$$

Como a definição de convolução de uma dimensão é de acordo com a equação 2.26 abaixo:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] h[n-k] \quad (2.26)$$

Portanto a equação 2.25 é uma convolução de uma dimensão, $\sum_{i=-\infty}^{\infty} x[m-i, n-j] h_1[i]$ e convoluída novamente com o *kernel* $h_2[j]$. Portanto uma convolução com um kernel de duas dimensões separável pode ser feita por duas convoluções separadas de uma dimensão [11]. O tempo de processamento de uma convolução de duas dimensões separada é menor que uma convolução de duas dimensões normal [11]. Terminada essa análise das convoluções separáveis explora-se o método de estimativa de fluxo ótico expansão polinomial.

2.2.4 Expansão Polinomial

Em resumo a expansão polinomial é uma aproximação de pixels adjacentes utilizando polinômios. Presume-se que o fluxo ótico é descrito por polinômios. Por exemplo, usando os polinômios quadráticos, obtem-se a equação 2.27:

$$f(x) \approx x^T A x + b^T x + c, \quad (2.27)$$

Em que A uma matriz simétrica, b um vetor e c um escalar [12]. Além disso, x é um vetor de coordenadas i, j, t , indicando coordenada horizontal, vertical e temporal da imagem respectivamente. Portanto x^T é o vetor transposto.

Os coeficientes são estimados através do método dos mínimos quadrados ponderados, *Weighted Least Squares* [12], de modo que correspondam ao sinal correspondente à sequência de frames. Utilizam-se dois componentes para essa estimativa: aplicabilidade e certeza. Após essa breve explicação do conceito da expansão polinomial e de onde deriva o método, utiliza-se o um cálculo do deslocamento, isto é um fluxo ótico em um contexto de imagem.

2.2.4.1 Estimativa de deslocamento

Começa-se a estimativa de deslocamento de uma sequência de imagens através da aproximação polinomial da Subseção 2.2.4. Define-se o próximo frame estimado por um deslocamento global d [12].

$$f_2(x) = f_1(x-d) = (x-d)^T A_1(x-d) + b_1^T(x-d) + c_1 \quad (2.28)$$

$$f_2(x) = x^T A_1 x + (b_1 - 2A_1 d)^T x + d^T A_1 d - b_1^T d + c_1 \quad (2.29)$$

$$f_2(x) = x^T A_2 x + b_2^T x + c_2 \quad (2.30)$$

Portanto conclui-se que:

$$A_2(x) = A_1(x), \quad (2.31)$$

$$b_2(x) = (b_1 - 2A_1 d), \quad (2.32)$$

$$c_2(x) = d^T A_1 d - b_1^T d + c_1 \quad (2.33)$$

Porém, presume-se A_1 não singular, tendo, o determinantes diferente 0. Portanto podemos encontrar o valor do deslocamento global d de acordo com as seguintes equações 2.34 2.35.

$$2A_1 d = -(b_2 - b_1) \quad (2.34)$$

$$d = \frac{1}{2} A_1^{-1} (b_2 - b_1) \quad (2.35)$$

Após definida como a expansão polinomial calcula o fluxo ótico, é necessário fazer correções que serão abordadas na próxima Subseção 2.2.4.2.

2.2.4.2 Correções Práticas

As suposições da Subseção 2.2.4.1 são aproximações que possuem um erro considerável, portanto são necessárias correções. Substitui-se a Equação 2.27 por expansão polinomial locais invés de um fluxo ótico global. Definimos os coeficientes de expansão $A_1(x)$, $b_1(x)$ e $c_1(x)$ para a primeira imagem e $A_2(x)$, $b_2(x)$ e $c_2(x)$ para a segunda. A aproximação deve resultar $A_1 = A_2$ de acordo com a equação 2.31, porém na prática utiliza-se a aproximação ilustrada pelas equações 2.36, 2.37 e o resultado 2.38 [12].

$$A(x) = \frac{A_1(x) + A_2(x)}{2} \quad (2.36)$$

$$\Delta b(x) = -\frac{1}{2}(b_2(x) - b_1(x)) \quad (2.37)$$

A relação que foi desenvolvida entre os coeficiente e deslocamento pode ser utilizado para um deslocamento variável. Portanto substituímos d , um deslocamento global, por $d(x)$, um deslocamento variável. A Equação 2.38 substitui as Equações 2.34 e 2.35, utilizando as novas condições:

$$A(x)d(x) = \Delta b(x) \quad (2.38)$$

Portanto para a estimativa entre dois pontos $x_1 = (i_1, j_1, t_1), x_2 = (i_2, j_2, t_2)$, que possuem coeficientes $A_1, B_1, C_1, A_2, B_2, C_2$, utiliza-se a equação 2.39 para calcular.

$$\frac{A_1(x) + A_2(x)}{2} d(x) = \frac{1}{2} (b_2(x) - b_1(x)) \quad (2.39)$$

Os resultados podem ser melhores, fazendo uma suposição de que o campo de fluxo está variando de forma gradual e lenta [12].

2.2.4.3 Estimativa pela vizinhança

Dado o problema da Subseção 2.2.4.2 minimizamos a disparidade dos resultados integrando os resultados vizinhos. Sendo Δx o deslocamento para se acessar a vizinhança do ponto x . Também define-se a função de peso $\omega(\Delta x)$ aos vizinhos. A função $\omega(\Delta x)$ corresponde a um fator de suavidade da estimativa de fluxo.

$$d(x) = \left(\sum \omega(\Delta x) A^T(x + \Delta x) A(x + \Delta x) \right)^{-1} \sum \omega(\Delta x) A^T(x + \Delta x) \Delta b(x + \Delta x), \quad (2.40)$$

Utilizamos ainda o valor $e(x)$ como valor de confiança, inversamente proporcional a confiança do fluxo calculado com fluxo vindo da expansão polinomial. Ilustra-se o calculo do valor $e(x)$ na equação 2.41 [12].

$$e(x) = \left(\sum \omega(\Delta x) \Delta b^T \Delta b(x + \Delta x) \right) - d(x)^T \sum \omega(\Delta x) A^T(x + \Delta x) \Delta b(x + \Delta x). \quad (2.41)$$

$e(x)$ é o valor de confiança de estimativa [12]. O valor de $e(x)$ pequeno indica uma estimativa de fluxo ótico de grande confiança.

Abordada a estimativa de fluxo ótico usando o fluxo da vizinhança descreve-se uma mensuração utilizada para a verificação e validação de algoritmos de fluxo ótico.

2.2.5 Endpoint Error

End-Point Error é uma métrica de erro absoluto utilizada em algoritmos de fluxo ótico [4]. Essa métrica pode ser considerada como o erro absoluto de uma medida vetorial do fluxo ótico [4]. Sua definição matemática é dada pela Equação 2.42.

$$e(x) = \sqrt{(u_0 - u_1)^2 + (v_0 - v_1)^2} \quad (2.42)$$

Sendo o vetor fluxo real $\vec{H}_0 = (u_0, v_0)$, o fluxo calculado $\vec{H}_1 = (u_1, v_1)$ [4] e o termo $e(x)$ o *End-Point Error*.

2.2.6 Performance de algoritmos de estimativa de fluxo ótico

Esta Seção apresenta um comparativo do nível de erro dos dois métodos de fluxo ótico apresentados, *Horn-Schunck* e *Expansão polinomial*. No contexto desse trabalho utiliza-se o tempo de processamento e o nível de erro como métrica de performance, pois parte-se da premissa do uso de um sistema embarcado com poucas capacidades computacionais. O Capítulo 4 explora o conceito de performance de algoritmos pelo tempo de execução.

Para uma avaliação do nível de erro dos algoritmos citados anteriormente utiliza-se o trabalho realizado pelo instituto *KiTTi* [5]. Nesse trabalho utiliza-se o processamento de imagens capturadas utilizando um carro em movimento. Em cima desse carro duas câmeras de alta resolução capturam imagens. Essas imagens capturadas diferentes são utilizadas para avaliação dos métodos e estão no Anexo II. A Tabela 2.1 apresenta o nível de erro da implementação do algoritmo *Horn & Schunck*.

Tabela 2.1: Erros da implementação Horn-Schunck [5]

Error	Fl-bg	Fl-fg	Fl-all
All / All	39.90 %	51.39 %	41.81 %
All / Est	39.90 %	51.39 %	41.81 %
Noc / All	30.49 %	48.25 %	33.71 %
Noc / Est	30.49 %	48.25 %	33.71 %

Na Tabela acima, 2.1, Fl-all é a porcentagem de fluxo ótico com erro considerando elementos no primeiro plano e no plano de fundo. Fl-bg é a porcentagem das disparidades calculadas apenas com regiões apenas consideradas como plano de fundo da imagem. Fl-fg é a porcentagem das disparidades calculadas apenas com regiões consideradas no plano da frente da imagem. O tempo de processamento da implementação foi de 2.6 minutos em um processador de 1 núcleo trabalhando em uma frequência de 3GHz [5] [13].

A análise do método de expansão polinomial, implementado usando a biblioteca aberta de Visão Computacional *OpenCV* [5], é apresentado na Tabela 2.2.

Tabela 2.2: Erros da implementação expansão polinomial [5]

Error	Fl-bg	Fl-fg	Fl-all
All / All	52.00 %	58.56 %	53.09 %
All / Est	52.00 %	58.56 %	53.09 %
Noc / All	43.77 %	55.90 %	45.97 %
Noc / Est	43.77 %	55.90 %	45.97 %

Seguindo a mesma lógica, *Fl-all* é a porcentagem de fluxo ótico com erros calculado no plano do fundo e primeiro plano. *Fl-bg* denominado a porcentagem das disparidades considerando apenas o plano de fundo da imagem. *Fl-fg* denominado a porcentagem das disparidades considerando apenas plano da frente da imagem. O processamento nas mesmas 20 imagens utilizadas na análise foi efetuado em apenas 1 segundo, em um hardware contendo processador de 1 core a 2.5 GHz e uma implementação em C/C++ utilizando usando OpenCV 2.1. Portanto pode se ver que o algoritmo de *Horn-Schunck* é mais preciso em sua estimativa. O algoritmo de expansão polinomial contém uma faixa de erros de 20% e uma pequena porcentagem a mais em estimativas de fluxo de regiões em primeiro plano. As comparações sobre esses dois algoritmos serão abordadas no Capítulo 4, abordando tempo de processamento. A característica de tempo de processamento é fator primordial para a interação da obra planejada. Após essa análise, finaliza-se os conceitos de fluxo óticos e começa a se adentrar sobre as opções de plataforma de desenvolvimento.

2.3 Plataformas de Desenvolvimento

Esta Seção introduz teoricamente as plataformas de desenvolvimento em Subseções. Este texto introduz e compara algumas das soluções do mercado disponíveis. As mais populares consistem nas linhas *Raspberry Pi*, *Arduino* e *Beaglebone*. A Subseção 2.3.1 irá apresentar a *Raspberry Pi* e as opções em mercado. A Subseção 2.3.2 explora o contexto da *Raspberry Pi* no mercado atualmente e após essa comparação será feita outra comparação com uma competidora no mercado de sistemas embarcados, *BeagleBone*, na Subseção 2.3.3.

2.3.1 Raspberry Pi

Raspberry Pi é uma linha de placas de desenvolvimento microprocessadas, não incluso a fonte de alimentação e o sistema operacional. Existem vários modelos disponíveis no mercado, tendo diferenças que são apresentadas na Tabela 2.3.

Tabela 2.3: Tabela de Modelos Raspberry Pi Disponíveis [6]

Modelo	Ram	USB Sockets	Ethernet Port	Notas
3B	1 GB	4	Sim	Inclui WiFi
Zero	512MB	1(Micro)	Não	Custo Baixo
2B	1GB	4	Sim	Quad-Core
A+	256MB	1	Não	
B+	512MB	4	Sim	Descontinuado
A	256MB	1	Não	Descontinuado
B rev2	512MB	2	Sim	Descontinuado
B rev1	256MB	2	Sim	Descontinuado

A Raspberry tem interface através dos seus pinos isolados eletricamente, a relação de nomes e numero dos pinos para serem utilizados constam a Figura I.2 em anexo.

Esse trabalho se concentrará em usar a versão *Raspberry Pi 2B*. Como a Tabela 2.3 mostra, esse modelo possui uma capacidade de processamento razoável, portanto seria a versão para o processamento para problemas de visão computacional propostos. Dada a introdução da ferramenta da Raspberry Pi, justifica-se o porquê de utilizar apenas a Raspberry e não o *Arduino*, uma ferramenta semelhante.

2.3.2 Comparação Entre *Arduino* e *Raspberry PI*

O dispositivo micro controlador *Arduino* é uma alternativa para o *Raspberry PI* no sentido de ser um micro controlador capaz de utilizar interfaces com periféricos através dos pinos digitais, portanto convém fazer uma diferença e uma comparação entre os dois e decidir qual seria mais indicado para o trabalho.

A primeira comparação a ser feita é em relação à interface humano-computador, o *Arduino* não possui interface de teclado, mouse ou tela. O *Raspberry Pi* por outro lado possui interfaces USB que permite a utilização de um mouse USB, teclado USB e um monitor via cabo HDMI [6].

As placas *Arduino* possuem quatorze (14) pinos digitais de input e output com corrente máxima de 40 mA, seis (6) pinos analógicos, 6 pinos PWM que são mais precisos que os pinos citados anteriormente. A *Raspberry Pi* possui até 26 pinos de input e output que suportam uma corrente máxima de apenas 3mA [6]. Placas *Arduinos* possuem uma memória de 32KB de flash e apenas 2KB. Comparando com o modelo mais fraco nessa categoria *Raspberry Pi* estaríamos comparando entre 256 MB de memória RAM 2.3. Processador da placa *Arduino* possui como frequência de relógio de apenas 16MHz. Compara-se com a frequência de relógio de 900MHz do *Raspberry Pi* [6]. Conclui-se que o *Raspberry* é mais indicado para problemas de alto

nível comparado com as Placas *Arduino*. As placas *Arduino* que possuem mais ferramentas para problemas de baixo nível, como motores servo. Comparados os dois produtos de sistemas embarcados, *Raspberry Pi* e *Arduino*, esse texto continuará com texto com comparação com outra opção, o *BeagleBone*.

2.3.3 Comparação entre Raspberry e BeagleBone

A linha de produtos de sistemas embarcados conhecida como *BeagleBone* compara-se com o *Raspberry Pi 2B* e o modelo *BeagleBone* que vai ser analisado junto é *Black*.

O *BeagleBone Black* dispõe da mesma capacidade memória que o *Raspberry Pi* então os dois podem ser utilizados nesse aspecto. Outro aspecto que pode ser comparado essas duas alternativas de sistemas embarcados o *Raspberry Pi* é necessária uma alimentação 5V e essa tensão pode ser obtida através de um cabo microUSB, já a alternativa do *BeagleBone Black* mantém essa necessidade de 5V mas não pode ser alimentado via USB. A memória RAM disponível nos dois produtos são comparáveis a *Raspberry Pi*, possuindo uma faixa de 256MB até 512MB e *BeagleBone Black* possui 512MB, no quesito flexibilidade para controle de periféricos, o *BeagleBone Black* possui uma vantagem em ter um conjunto de pinos analógicos que o *Raspberry Pi* não possui. A comunicação principal do *BeagleBone Black* e *Raspberry Pi* são iguais pelo port RJ45 Ethernet com velocidades de conexão iguais. Os sistemas operacionais que controlam a *Raspberry Pi* e *BeagleBone Black* são, respectivamente, uma distribuição do Debian chamada Raspbian, que possui uma comunidade bem disseminada, e o Linux Angstrom, que é uma distribuição baseada no Linux sendo um sistema operacional bem leve e com características minimalistas [14]. Portanto a conclusão desse comparativo é que tanto a placa *BeagleBone Black* e *Raspberry* são alternativas viáveis, mas a mais segura devido a sua comunidade e facilidade de utilização a escolha para esse trabalho entre as duas seria a *Raspberry*.

Após ser abordado detalhes das principais plataformas de desenvolvimento, explora-se a linguagem e ambiente de programação *Processing*.

2.4 Processing

Essa Seção descreve a ferramenta de programação utilizada nesse trabalho e suas funcionalidades. Primeiramente, a Subseção 2.4.1 irá ilustrar e definir a ferramenta, logo após será exemplificado o funcionamento da biblioteca *io* do *Processing* na Subseção 2.4.2.

2.4.1 Definição e Aplicações

Processing é um software de código aberto ambiente e linguagem de programação no contexto de artes visuais. A linguagem *Processing* é implementada em *Java*, *Python*, *Javascript*, portanto

utiliza os recursos das linguagens as quais foram implementadas. Portanto a linguagem *Processing* possui várias versões. A linguagem possui ferramentas nativas para desenho além de a linguagem ter uma linha de execução em loop, o que é perfeito para animações [15]. O domínio de aplicação pode ser ilustrado não somente na descrição mas também em algumas iniciativas pela comunidade do *Processing*. Apresenta-se alguns trabalhos que são de código aberto que são compartilhados e apresentados no site OpenProcessing. Um detalhe importante em relação a esses trabalhos é que eles são produtos da utilização de uma versão portátil para a sintaxe da linguagem javascript [16].

Terminada a introdução do domínio de aplicação da ferramenta do *Processing* e sua definição, porém é necessário para execução desse trabalho a utilização de funcionalidades para acessar periféricos em geral, portanto será explorada a biblioteca de hardware padrão do *Processing*.

2.4.2 Biblioteca Hardware

A biblioteca de input e output, *io*, permite acesso aos periféricos hardware, como inputs e outputs digitais e *serial busses*. Essa biblioteca é destinada para uso em sistemas embarcados baseados em Linux com os kernels necessários, como a *Raspberry Pi* [17].

A biblioteca de hardware possui a classe GPIO que possui como funcionamento utilizar pinos considerados GPIOs do sistema embarcado. A classe GPIO possui a funcionalidade de configurar os pinos de modo a serem considerados como saída ou entrada de sinais. Por exemplo configuração de pinos de entrada e saída é implementada através do método *pinMode()*. A funcionalidade de colocar o valor booleano em um dos pinos digitais GPIOs já configurados em modo de saída é implementada através do método *digitalWrite()*. As funções descritas podem ser chamadas de acordo com o seguinte exemplo de código em *Processing*, o listing 2.1 :

Algoritmo 2.1: Exemplo de uso do método digitalWrite() [18]

```

1 import processing.io.*;
2 boolean ledOn = false;
3
4 void setup() {
5   GPIO.pinMode(4, GPIO.OUTPUT);
6
7   // On the Raspberry Pi, GPIO 4 is pin 7 on the pin header,
8   // located on the fourth row, above one of the ground pins
9
10  frameRate(0.5);
11 }
12
13 void draw() {
14   ledOn = !ledOn;
15   if (ledOn) {
16     GPIO.digitalWrite(4, GPIO.LOW);
17     fill(204);
18   } else {
19     GPIO.digitalWrite(4, GPIO.HIGH);
20     fill(255);
21   }
22   stroke(255);
23   ellipse(width/2, height/2, width*0.75, height*0.75);
24 }

```

Como visto no exemplo de código, Algoritmo 2.1, há instâncias de utilização das funções da biblioteca da *io*, Hardware, do *Processing*. Na linha 5 do código está escrito *GPIO.pinMode(4,GPIO.OUTPUT)*, sendo a classe *GPIO* uma classe estática no contexto de orientação de objeto, o número inteiro 4 indica qual *GPIO* esta sendo configurando. Um exemplo de utilização da configuração do pino é configurar um pino digital na *Raspberry Pi* modelo 2B por exemplo, segundo a Tabela I.2 indica que no exemplo se configura o pino de numero 7 na Tabela e o valor *GPIO.OUTPUT*. A constante estática *GPIO.OUTPUT* indica qual papel esse pino vai desempenhar entre duas opções disponíveis, leitura de sinal ou emitir o sinal. Na linha 16 e 19 está escrito *GPIO.digitalWrite(4, GPIO.HIGH);*, novamente é utilização de um método estático no contexto de orientação a objeto. O número 4 como parâmetro indica novamente qual *GPIO* será a utilizada para emissão do sinal. O valor *GPIO.HIGH* é um valor interno a classe indicando

uma voltagem que pode ser compreendida como um valor lógico *VERDADEIRO* e respectivamente o valor *GPIO.LOW* com o valor falso. Após ter sido ilustrado o funcionamento da biblioteca *io* será apresentado a biblioteca de visão computacional e sua instância em específica para o *Processing*.

2.5 OpenCV

OpenCV é a biblioteca de código aberto de visão computacional e *machine learning* [19]. A biblioteca possui mais de 2500 algoritmos otimizados para visão computacional e machine learning [19]. O uso da biblioteca pode ser medido através do número de downloads, que atualmente excede 14 milhões [19]. A biblioteca *opencv* possui interfaces em várias linguagens para serem utilizadas com as ferramentas correspondentes, por exemplo, C++ , C, Python, Java, MATLAB [19]. Devido a portabilidade para java, foi possível criar uma biblioteca *opencv* baseada na biblioteca *OpenCV java*, atualmente na versão 2.4.5. A biblioteca implementada para *Processing* implementa uma série de classes que possuem métodos que são implementações de algoritmos de visão computacional apenas. A Figura 2.4 exemplifica a utilização da biblioteca OpenCV para *Processing* através de um código apresentado no Listing 2.2 [20].



Figura 2.4: Exemplo OpenCV

O Algoritmo 2.2 é a implementação de um exemplo de estimativa de fluxo ótico. O programa reproduz um video previamente gravado e estima o fluxo ótico do video em tempo real. A representação do fluxo ótico estimado é um conjunto de setas vermelhas, o fluxo ótico médio estimado é representado por uma reta cinza, mostrado na Figura 2.4.

Na linha 10 do Listing 2.2 instancia-se o objeto que abstrai os algoritmos e funcionamento da *opencv*, utilizando a palavra reservada *this*, que indica a instância de um objeto *sketch* do *Processing*. O objeto *sketch* pode ser entendido como a abstração do programa como um todo. Os parâmetros 568 e 320 indicam o tamanho dos dados a serem processados, isto é, as imagens nesse caso a serem processadas são 568 pixels de largura e 320 de altura.

Observa-se também que na linha 19, o método *loadImage(video)* carrega o objeto *Movie* video no contexto do objeto *Opencv*, sendo o objeto *Movie* uma abstração de uma sequência de *PImage*'s que são retornadas toda vez que um frame novo está disponível. Outro detalhe importante a ser descrito são os métodos *calculateOpticalFlow()* e *drawOpticalFlow()* respectivamente o método que usa o algoritmo de cálculo do fluxo ótico da *Opencv* e o método que desenha pequenos vetores correspondentes ao fluxo ótico de cada pixel independentemente.

O algoritmo implementado no *calculateOpticalFlow()* utiliza o método *farneback*, utilizando a implementação da API *Opencv* de java. Após cálculo do fluxo ótico pode ser obtida a matriz resultado ou partes dessa mesma matriz com os métodos *getAverageFlowInRegion(int x, int y, int w, int h)* e *getFlowAt(int x, int y)*, sendo (x,y) a posição na imagem em pixels, w a largura da secção da imagem, e h a altura da secção da imagem.

Algoritmo 2.2: "Exemplo minimalista de aplicação da Opencv e fluxo ótico"

```
1 import gab.opencv.*;
2 import processing.video.*;
3
4 OpenCV opencv;
5 Movie video;
6
7 void setup() {
8     size(1136, 320);
9     video = new Movie(this, "sample1.mov");
10    opencv = new OpenCV(this, 568, 320);
11    video.loop();
12    video.play();
13
14 }
15
16 void draw() {
17     background(0);
18     if(video.width > 0 && video.height > 0){
19         opencv.loadImage(video);
20         opencv.calculateOpticalFlow();
21         image(video, 0, 0);
22         translate(video.width, 0);
23         stroke(255, 0, 0);
24         opencv.drawOpticalFlow();
25     }
26 }
27
28 void movieEvent(Movie m) {
29     m.read();
30 }
```

Explorado a linguagem e ambiente programação *Processing* aborda-se o uso de placas de circuito impresso ou mais conhecidas como PCIs ou PCBs

2.6 Placas de Circuito Impresso (PCB) e circuitos integrados

Esta seção consiste de subseções de definição em 4(quatro) partes 2.6.1 e descrição das camadas que define as PCI ou PCB, 2.6.2 e 2.6.3 sendo respectivamente camadas diferentes sendo definidas e exploradas em cada Subseção. A Subseção 2.6.4 informa o funcionamento do *shift register*, o circuito integrado que paraleliza uma entrada serial.

2.6.1 Definição

O objetivo da placas de circuito impresso é a criação de ligações elétricas através de trilhas, evitando a necessidade do uso de fios condutores em um circuito.

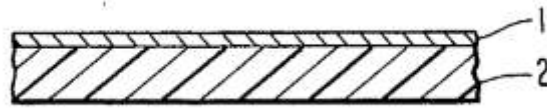


Figura 2.5: Esquema Conceitual de uma Placa de Circuito Impresso - PCB. 1 indica a camada A e 2 indica a camada B

A Figura 2.5 representa as camada A e camada B, as camadas são indicadas pelos números 1 e 2 respectivamente. Uma placa de circuito impresso é um conjunto de camadas físicas que possuem determinadas funções, descreve-se uma a uma suas camadas.

2.6.2 Camada A

A camada A consiste de um corpo em forma de uma parede fina de um composto. Esse composto consiste de uma mistura de *polímero de etileno/comonômero A* e *polímero etileno/comonômero B* com opção de uma terceiro composto de um elemento inorgânico condutor de calor. Sendo o *polímero etileno/comonômero A* consistindo em etileno e um composto epóxido possuindo de 6 até 30 átomos de carbono e possuindo pelo menos uma ligação dupla. O *polímero etileno/comonômero B* consiste de um polímero composto de etileno e e epóxido contendo até 25 átomos de carbono, ácido acrílico, ácido metacrílico e outros ácidos [21]. Após descrita a primeira camada, descreve-se ia camada B ou a camada de metal condutor a eletricidade

2.6.3 Camada de metal condutor

A camada de de metal pode ser obtida a partir de uma deposição do metal, e podem ser de diferentes espessuras, variando de 100 Å até 400 μm , com condutividade entre 10^{-7} até 0.2Ω .

2.6.4 Shift Register 745HC595

O circuito integrado 745HC595 *shift register* paraleliza um sinal em série. O circuito integrado é ilustrado na Figura 2.6:

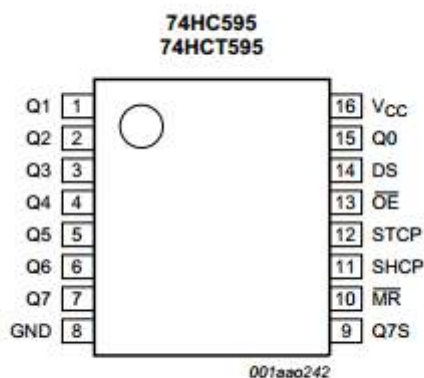


Figura 2.6: Shift register 745HC595

A pinagem do circuito integrado é mostrada na Tabela 2.4.

Tabela 2.4: Pinagem do Shift register 74HC595

Simbolos	Pinos	Descrição
Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7	15, 1,2,3,4,5,6,7	Data output paralelo
GND	8	Ground (0 V)
Q7'	9	Carry out
MR	10	Reset master(Ativo no Gnd)
SHCP	11	Input clock do registrador shift
STCP	12	Input clock do registrador storage
OE	13	Habilitador output
DS	14	Input data serial
Q0	15	Output paralelo 0 (vai até a Q7)
Vcc	16	Fonte de voltagem

Os sinais de entrada do circuito integrado da Figura 2.6 e na Tabela 2.4 são compostos de três sinais paralelos. O primeiro dos três sinais paralelos foi utilizado para enviar sequências de bits, chamado de *DS* ou *SER*. O segundo sinal paralelo emite o sinal de clock de shift do shift register 745HC595, o pino correspondente na Tabela 2.4 chama-se SHCP O terceiro sinal paralelo emite o sinal de clock de store do shift register 745HC595, o pino correspondente na Tabela 2.4 chama-se STCP. Os sinais de configuração do circuito integrado são compostos de 8

sinais isolados. Os pinos *GND* e *Vcc* são o *Ground* e a fonte de alimentação do circuito respectivamente. O pino *MR* controla o comportamento de limpar a memória do circuito apagando as entradas anteriores. O comportamento de apagar a memória do circuito é ativado colocando o valor lógico falso no pino *MR*. O pino *OE* controla o bloqueio da saídas do circuito, caso esteja recebendo o sinal lógico falso o circuito está bloqueado. Caso esteja bloqueado o valor lógico nos pinos *Q1, Q2, Q3, Q4, Q5, Q6, Q7* e *Q7'* são sempre valores lógico falsos. O sinal de saída é composto de 9 (nove) pinos de saída, *Q1, Q2, Q3, Q4, Q5, Q6, Q7* e *Q7'*. Cada pino corresponde a uma posição na memória interna do circuito. O circuito possui uma memória em pilha, a medida que a entrada *SH_CLK* vai recebendo um *clock* a memória armazena um valor lógico. O valor lógico armazenado é o atual valor no pino *SER*. A saída do circuito são os valores que estão na memória no instante em que o pino *STR_CLK* recebe uma borda de subida. A memória possui tamanho para oito (8) valores lógicos por vez. Quando tenta armazenar um nono valor o circuito retira o último valor da pilha e o coloca no pino *Q7'*. Após retirar o primeiro valor da pilha o circuito coloca o segundo valor na primeira posição, depois coloca terceiro no lugar do segundo. Dessa forma o novo oitavo valor é o nono valor que tenta-se armazenar [22].

2.7 Trabalhos Relacionados

Esta seção trabalhos relacionados mostra um estudo de vários trabalhos semelhantes já feitos. A Tabela 2.5 ilustra vários trabalhos relacionados.

Tabela 2.5: Lista de trabalhos utilizando visão computacional e placas de desenvolvimento.

Trabalho	Hardware	Software
Priyanka G. <i>et al</i> [23]	Intel core 2 Duo	MATLAB (R2013a 8.1v)
Wilson F. [24]	RPI B.	(Não especificado)
Yukihiro S. [25]	RPI B.	Python/OpenCv
Iszaidy <i>et al</i> [26]	RPI B+.	C/C++ /Opencv
Yi-You Hou [27]	RPI (Não especificado).	Visual Studio C# / Opencv Opengl
Senthil Kumar <i>et al</i> [28]	RPI 3.	(Não especificado)
Pawel S. <i>et al</i> [29]	RPI B.	C/C++ & Python (Encasing) / Opencv

O presente trabalho tem precedentes de outros projetos já feitos e escritos. Na Tabela 2.5 temos oito(8) trabalhos que utilizam a plataforma de desenvolvimento da linha *Raspberry Pi*, a única exceção é encontrada no trabalho *Priyanka G. et al* que utiliza majoritariamente um desktop que dispõe de um processador diferente dos que temos nas plataformas de desenvolvimento. Esse trabalho em específico usa o processador do gabinete para fazer o processamento. Esses dados processados são transmitidos para a plataforma de desenvolvimento continuar com a execução. A Tabela 2.6 lista os algoritmos usados de visão computacional.

Tabela 2.6: Lista de trabalhos e quais algoritmos foram usados.

Trabalho	Algoritmo
Priyanka G. <i>et al</i> [23]	Lucas-Kanade
Wilson F. [24]	Human Detection/Smoke Detection
Optical Flow Measurement System.	Horn & Schunck /Lucas-Kanade
Yukihiro S. [25]	Não especificado.
Iszaidy <i>et al</i> [26]	Trabalho
Yi-You Hou [27]	Horn Schunck
Senthil Kumar <i>et al</i> [28]	Lucas-Kanade

Na Tabela 2.6, o primeiro trabalho utiliza o algoritmo *Lucas-Kanade*, em segundo lugar o está o algoritmo *Horn-Schunck*. Os outros trabalhos que utilizam um algoritmo como *Human Detection, Tracking and Travel Time e Real-time Detection and Tracking* não são de fluxo ótico. Escrito isso, finaliza-se o Capítulo 2 de fundamentação teórica. O próximo Capítulo é o de Materiais e Métodos e explica todo o processo de desenvolvimento deste trabalho.

Capítulo 3

Materiais e Métodos

O presente trabalho consiste de uma obra de arte computacional generativa, a qual é ilustrada na Figura 3.1 Aproxima-se o público a ciência da computação e uma forma de vida computacional. A obra consiste de uma matriz de *LEDs* de 5 colunas por 10 linhas. Essa matriz de *LEDs* é controlada por um *software* na plataforma de desenvolvimento *Raspberry Pi* por sinais passados pelos pinos de propósito geral. Os sinais passados da *Raspberry Pi* são pré-processados em uma placa de circuito impresso (PCB), a qual separa o sinal em 50 sinais paralelos. Esses sinais paralelos fecham o circuito dos *LEDs* ligando-os. A *Raspberry Pi* está ligada através de uma interface *USB* a uma *webcam*. Os *LEDs* representam uma célula com formato quadrado que são controlados pela *Raspberry Pi* e possuem o comportamento de vagar aleatoriamente e perseguir 'alimentos' nessa matriz de *LEDs*. O *software* transforma movimento captados pela *webcam* em 'alimentos', os algoritmos de fluxo ótico estimam o movimento.

Nesse capítulo descrevemos o projeto construído, com detalhamento científico de forma que seja repetido para uma eventual próxima iteração de outros artigos científicos com mesmo intuito de explorar a arte e tecnologia. O processo deste trabalho de conclusão de curso foi dividido em quatro Seções. A primeira seção, 3.1, explica como foi realizada a construção da obra em relação a placas de circuito impresso e configuração da plataforma de desenvolvimento. A Seção 3.2, consiste a explanação do desenvolvimento do *software* e sua interface com o hardware da obra. A Seção 3.3 fala o que foi feito para a construção da estrutura de madeira que engloba a obra.



Figura 3.1: Foto da obra sendo testada.

3.1 *Hardware*

Esta seção consiste de descrever o que foi feito na área de hardware na construção da obra de arte eletrônica. A Subseção 3.1.1 aborda como foi projetado e implementada a parte específica do *Raspberry* e toda configuração do sistema operacional *Raspbian*. A Subseção 3.1.2 apresentada o processo de fabricação de placa de circuito impresso.

3.1.1 Raspberry Pi

O *Raspberry* é a plataforma utilizada para executar o software e usar os pinos GPIO padrão da *raspberry* para comunicação com hardware. A versão do raspberrry utilizada é o modelo Raspberry 2 B + [30], operada com o sistema operacional *raspbian*, mostrada na Figura 3.2.



Figura 3.2: Raspberry Modelo 2B

O sistema operacional foi preparado para executar automaticamente um programa ao efetuar a operação de boot no cliente default chamado *pi*. Através da criação de um arquivo chamado *auraTerminalPopup.desktop* na pasta `/home/pi/.config/autostart/` o conteúdo do arquivo é descrito no Listing 3.1:

Algoritmo 3.1: Código autostart

```
1 [Desktop Entry]
2 Exec=lxterminal -e "<PATH_PARA_EXECUTAVEL> &"
3 Type=application
```

Colocado esse arquivo não é mais necessário o uso de teclados para rodar um arquivo executável, sendo apenas necessário colocar o *raspberry* na fonte correta. Após configuração da execução do programa de forma automática é necessário configurar a *webcam* utilizada pelo *Raspberry Pi*. Utiliza-se webcam do modelo *Life Cam VX-2000* e infelizmente não há uma

driver plug and play para o sistema operacional *Raspberry Pi* portanto foi utilizado o pacote de drivers chamado *V4l2, video for linux 2*, e utiliza-se os comandos do programa listado no Listing 3.2.

Algoritmo 3.2: Configuração da webcam

```

1 [ Desktop Entry ]
2 v4l2-ctl --info
3 v4l2-ctl --list-ctrls
4 v4l2-ctl --set-ctrl white_temperature_balance_auto=0

```

A linha 1 indica como receber todas as informações da *webcam* instalada. A linha 2 é o comando de informar uma lista de parâmetros de controle da câmera que podem ser alterados e na linha de número 3 altera-se o parâmetro "white_temperature_balance_auto" para o valor falso, isto é, 0. Essas configurações foram executadas uma única vez. O *Raspberry Pi* mantém essas configurações mesmo se houver desligamentos ou o reiniciar da plataforma de desenvolvimento.

3.1.2 PCB

Esta Subseção será descritiva em relação a todo processo de fabricação da placa de circuito impresso. Os materiais necessários para se criar a placa de circuito impresso são apresentado na Tabela 3.1 e irá ser usado como base:

Tabela 3.1: Lista de peças para Fabricação da placa circuito Impresso

Modelo	Quantidade
Conectores barra de pinos fêmea (1x3x11,2-180°)	50
Soquete circuito integrado 16 pinos	7
Conector barra Pinos Macho (1x10x11,2-180°)	1
Conector barra de pino fêmea (1x6x11,2-180°)	1
Shift register 74hc595	7
CONNECTOR BORNE KRE- 2T (PASSO 5MM - AZUL)	25
Tip 120 NPN	50
Micro retífica FORT FT 4510	1
Ferro de Passar Roupa	1
Folha Papel Couchê(Adesivo) Transparente	1
Impressora de Jato de Tinta	1
Placa de Cobre 20(cm)x20(cm)	1
Percloroeto de Ferro (Gramas)	1

A placa de circuito impresso foi utilizada para ser possível o uso de apenas três (3) pinos GPIO para determinar o estado individual de cada fita de LED.

O esquema da placa impressa conceitualmente é demonstrado na Figura 3.3:

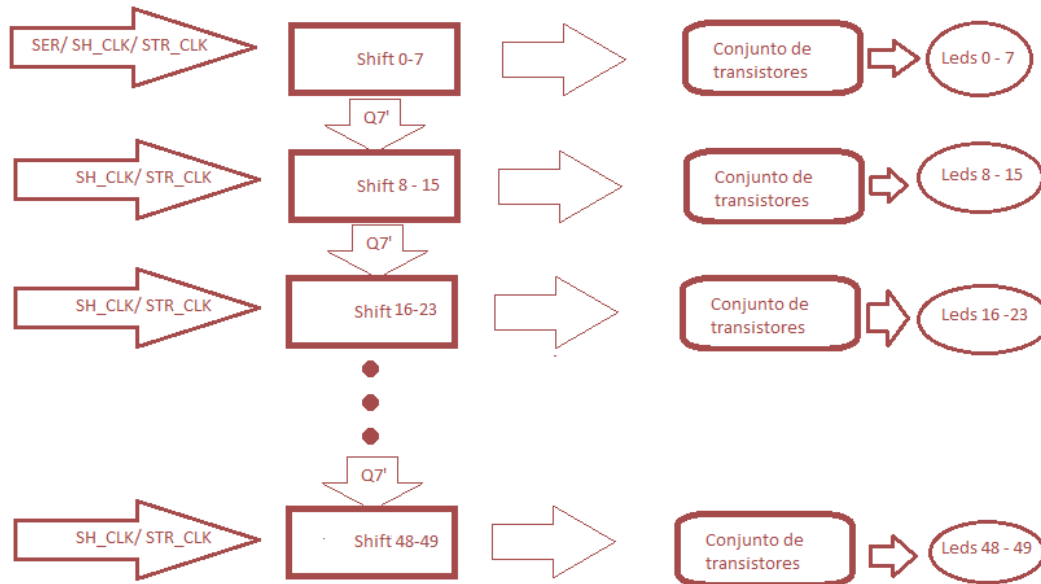


Figura 3.3: Esquemático lógico simplificado da PCB, original em anexo I.1

A Figura 3.3 resume o funcionamento que a placa de circuito impresso. O primeiro *shift register* recebe o sinal *SER*, *SH_CLK* e *STR_CLK*. O sinal *SER* são os dados em serial, o sinal *SH_CLK* separa os bits dos dados seriais e o sinal *STR_CLK* sinaliza o *shift register* para descarregar os dados paralelizados. Cada *Q7'* indica o *carry* de cada *shift register* para o próximo *shift register*, essa característica é o que possibilita o encadramento dos *shifts*. O conjunto de transistores são ativados pelas saídas *Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7* de cada *shift register*. O conjunto de transistores são correspondentes aos *LEDs* 0 a 7, cada transistor controla um *LED*. A produção da placa de circuito impresso foi realizada através do software de design de circuito *eagle* [31]. Seguindo o processo de produção da placa impressa, o software *eagle* converte o esquemático para o desenho da placa impressa, o resultado é o mostrado 3.4:

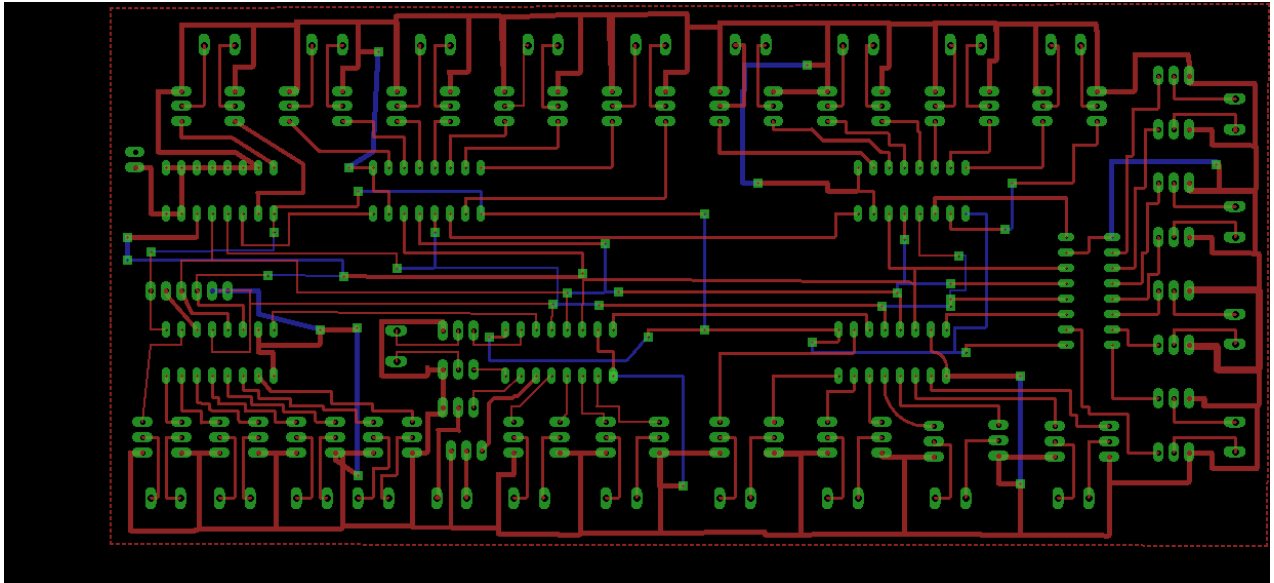


Figura 3.4: Esquema da PCB

No esquema da impressão da placa, 3.4, os fios vermelhos são da camada superior da placa, os fios azuis são da camada inferior, os contatos verdes são pontos onde serão soldados os circuitos e os contatos entre as camadas superior e inferior. A placa mede 20 cm por 10 cm e a camada superior da placa é composta de cobre e um material plástico isolante.

O projeto da placa foi feito através de um processamento feito no próprio *eagle*. O processamento feito pelo *eagle* é chamado de *netstat* e faz o isolamento no desenho da placa de todos os fios. Foi criada um arquivo em formato *.jpg* mostrado na Figura 3.5 aplicando um processo de negativar as cores das imagem anterior e rotacionar de horizontalmente.

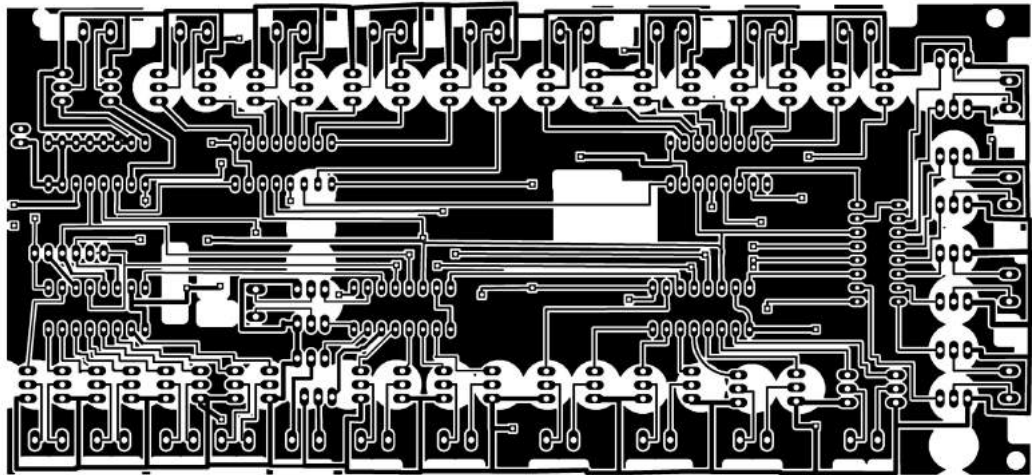


Figura 3.5: Imagem do Projeto invertida horizontalmente preenchida e negativada

O primeiro passo para a criação é imprimir a imagem em uma papel couchê adesivo transparente por uma impressora de jato de tinta. Utiliza-se uma impressora de jato de tinta pois em um papel transparente as impressoras de laser não conseguem imprimir de forma eficaz. Depois prensa-se com uma chapa quente o papel couchê e a placa de cobre. O papel couchê necessita estar com o lado da tinta virado para a placa de cobre. Utiliza-se um pano para proteger a placa de queimaduras, que no caso desse trabalho foi utilizado uma ferro de passar roupa para prensar a impressão na placa de cobre.



Figura 3.6: Prensando o papel couchê na placa de cobre

Depois de feito a prensa coloca-se a mistura de percloroeto de ferro e água em um recipiente de plástico e mergulhamos a placa nessa solução. A solução corrói apenas as partes onde não há tinta. Esse processo cria uma placa de cobre de acordo com o projeto criado no software *Eagle*. É necessário manter uma forma como manusear a placa imerso na solução não encostando com a mão, uma fita crepe adesiva grudada em uma parte onde não existe marcações da tinta. As imagens ilustrando esse processo estão abaixo:



Figura 3.7: Banho de percloroeto de ferro

Logo após esse procedimento foi utilizado a micro retífica para furarmos onde os circuitos integrados iriam ser soldados com estanho.

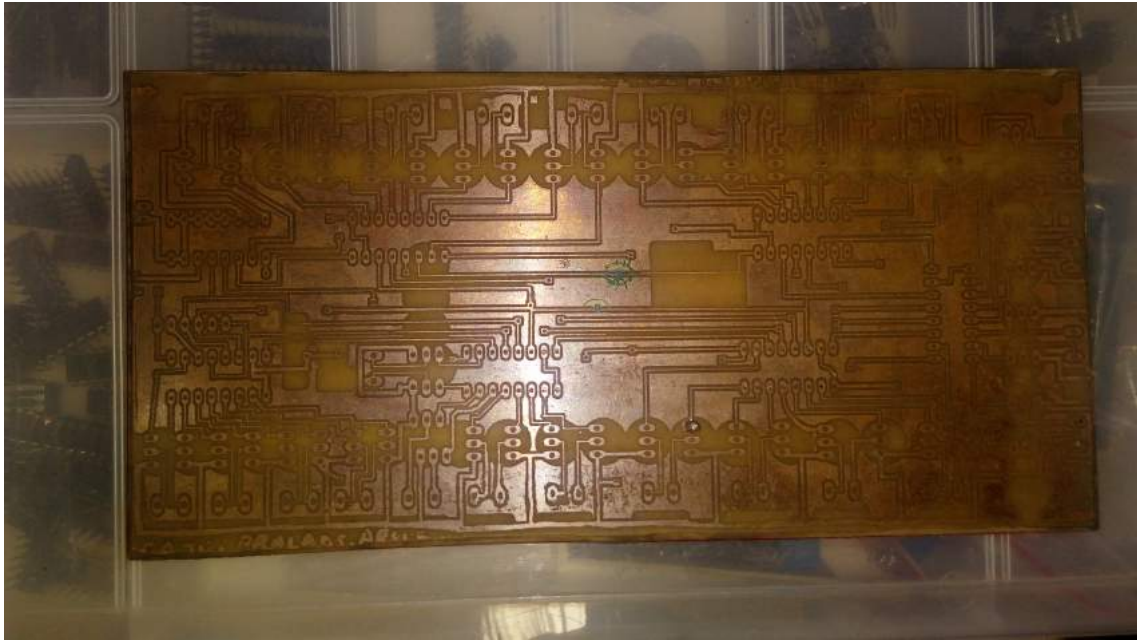


Figura 3.9: Resultado do processo do banho de percloroeto de ferro.



Figura 3.8: Utilização da micro retífica na placa

Depois do banho de percloroeto a placa de cobre fica da forma ilustrada na Figura 3.9.

Feito os passos anteriores foram soldados os 50 conectores fêmeas de 3 pinos em cada furos triplos entre os conjuntos de 16 pinos destinados ao soquete 3.1 e os furos destinado para os conectores BORNE. Os conectores fêmeas de 3 pinos são destinados para inserção dos Tips. Solda-se o soquete de 16 pinos e o conector fêmea de 16 pinos nos lugares indicados pela Figura

abaixo:

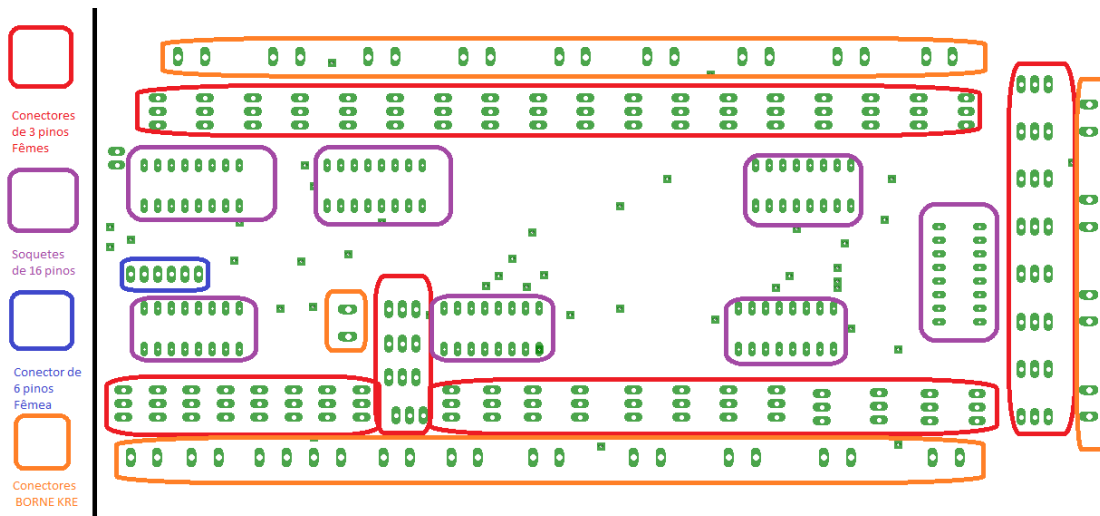


Figura 3.10: Soldagem dos componentes

O resultado da soldagem de todos os componentes nos lugares indicados na Figura 3.10 estão nas Figuras 3.11 e 3.12.

As Figuras 3.11 e 3.12 ilustram a placa de circuito impresso após soldagem de todos os componentes. Porém o projeto ilustrado na Figura 3.4 possui um erro no valor lógico do pino *MR*, portanto foi necessário fazer um fio aéreo que liga todos os pinos de *MR* dos *shift registers*. Esse fio aéreo foi colocado o valor lógico verdadeiro. A corrosão da placa foi falha em alguns pontos, portanto foram soldados fios aéreos onde ocorreu falha. O resultado de todas essas correções estão ilustradas nas Figuras 3.11 e 3.12.



Figura 3.11: Camada Inferior após correções

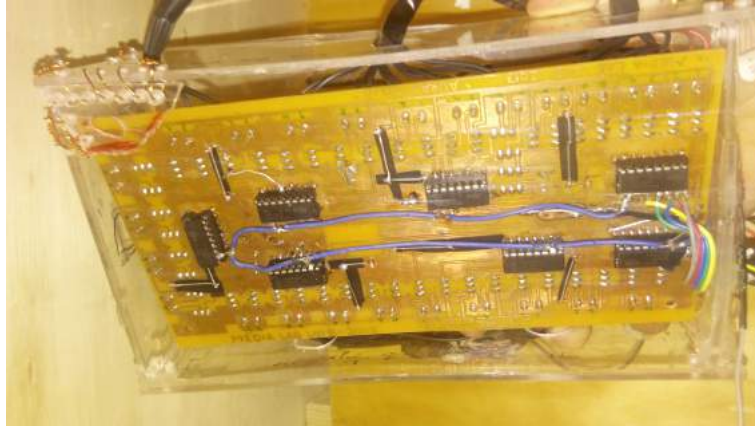


Figura 3.12: Camada Superior após correções

Finalizado o processo de fabricação da placa de circuito impresso explora o processo feito na parte de *software*.

3.2 Software

Esta subseção consiste do detalhamento do software utilizado no *Raspberry Pi*. Para intuito de comparação foi implementado duas versões da estimativa de fluxo ótico. Essas versões e suas características são explicadas posteriormente nesse capítulo. O programa tem como responsabilidade controlar o comportamento das células, emitir sinais para os *LEDs* e estimar o movimento captado pela *webcam*. O programa final é dividido em quatro (4) módulos, sendo o módulo *protótipo* o principal que importa e instancia objetos dos outros módulos, *Cell*, *Camera*, *InterfaceHardware*. As subseções seguintes descrevem cada módulo, suas responsabilidades e sua estrutura separadamente. Esta seção consiste de explicações que introduzem as classes utilizadas na etapa de software desse trabalho, as Subseções 3.2.1 , 3.2.2, 3.2.3 e 3.2.4 , "Comportamento Celular", "Captura de Imagem", "InterfacedeHardware" e "Implementações de estimativa de fluxo ótico "respectivamente. Primeiramente se ilustra as três implementações nas Figuras 3.13, 3.14 e 3.15.

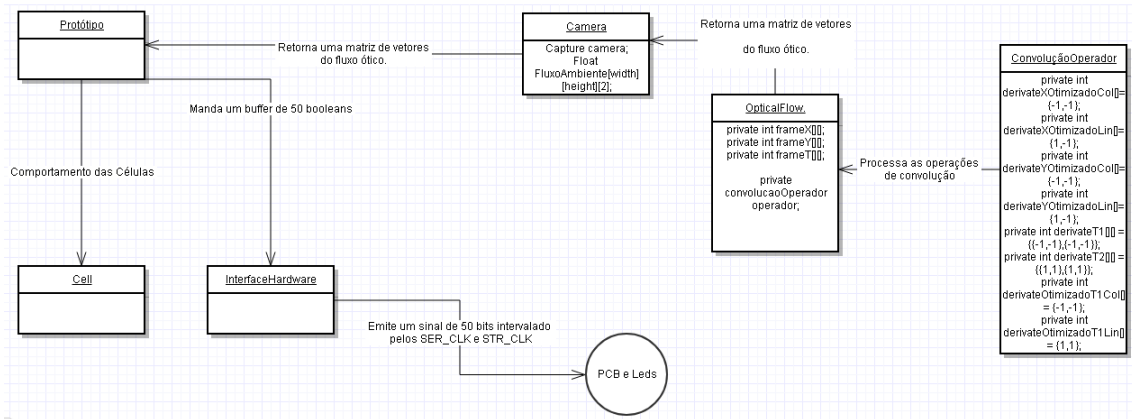


Figura 3.13: Software Horn & Schunck Serial.

A Figura 3.13 ilustra todas as classes utilizadas nessa implementação. A classe principal chama-se *Prototipo* e possui responsabilidade de controlar e utilizar todas as outras classes. A classe *Cell* possui a responsabilidade de implementar o comportamento celular. A classe *InterfaceHardware* possui a responsabilidade de emitir um sinal que representa o ambiente celular para os circuitos externos à Raspberry Pi. A classe *Camera* possui responsabilidade de capturar imagens utilizando a *webcam* e utilizar as classes de estimativa de fluxo ótico. No caso da implementação do *Horn & Schunck* serial as classes que estimam o fluxo ótico são as classes *OpticalFlow* e *convolucaoOperador*. A classe *OpticalFlow* implementa o método *Horn & Schunck*, as convoluções são calculadas utilizando a classe *convolucaoOperador*. A segunda implementação que utiliza a expansão polinomial com a *OpenCV* e é ilustrada na Figura 3.14.

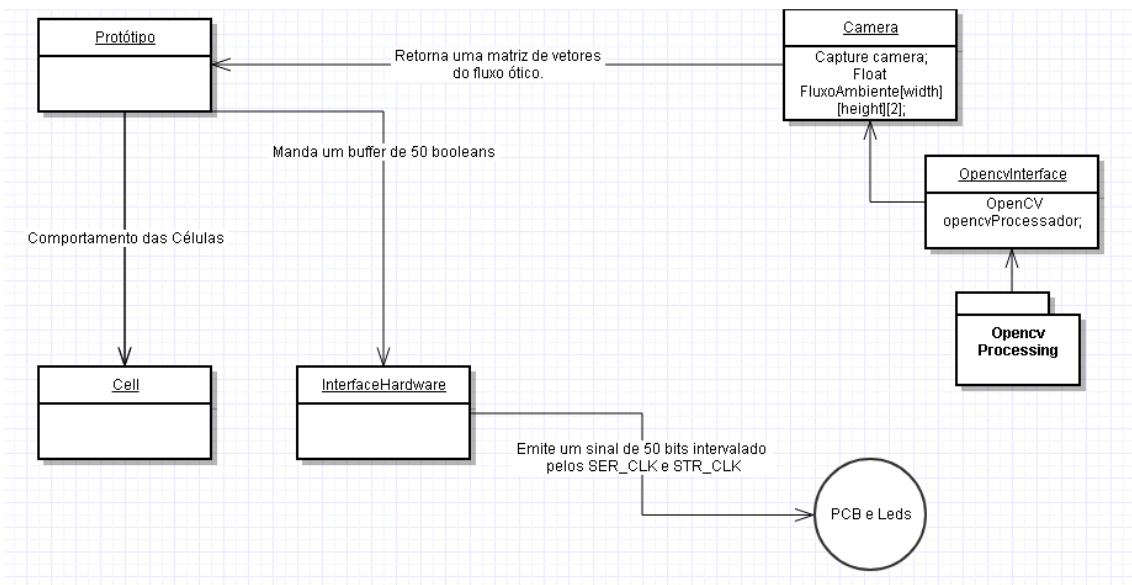


Figura 3.14: Software expansão polinomial/OpenCV.

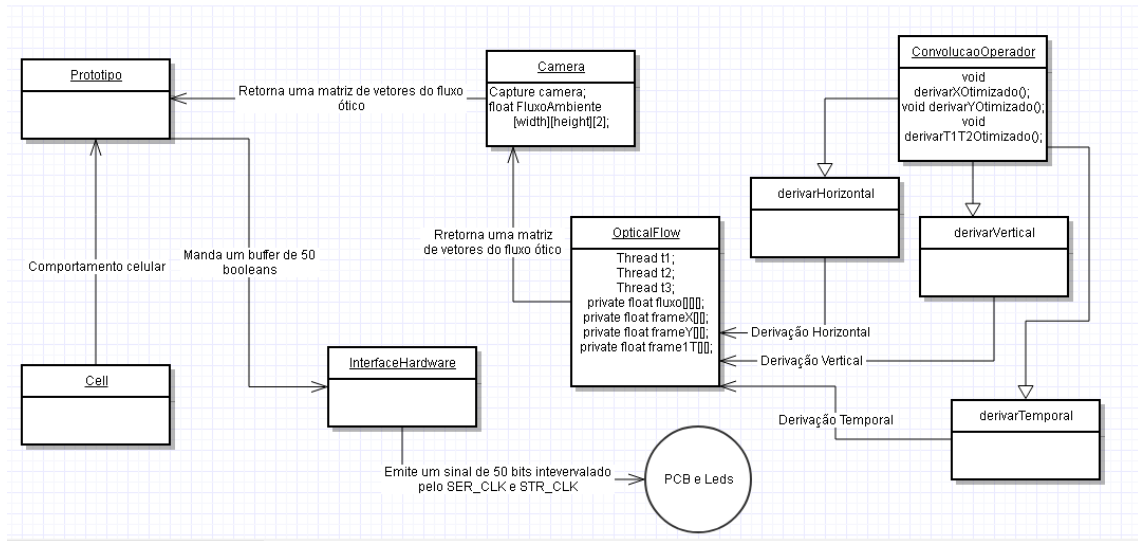


Figura 3.15: Software Horn & Schunck paralelo.

A segunda implementação na Figura 3.14 tem poucas diferenças. A classe que estima o fluxo óptico é a classe *OpenCvInterface* que utiliza as ferramentas da biblioteca *OpenCV*. A versão da biblioteca é a versão em *Processing*. Ilustra-se a terceira implementação utilizando o método *Horn & Schunck* na Figura 3.15. A Figura 3.15 difere da primeira implementação do Horn & Schunck incluindo classes auxiliares. As classes auxiliares *derivarHorizontal*, *derivarVertical* e *derivarTemporal* são instanciadas como *Threads* paralelas. As convoluções necessárias para o método *Horn & Schunck* são feitas por essas *Threads* e são instanciadas pela classe *OpticalFlow*.

A enumeração *Direcao* é utilizada em todo o contexto do programa. Essa enumeração implementa as direções em que as células podem se movimentar.

- Enumeração *Direcao*:

A enumeração definida no arquivo e na classe *Cell.pde* e é utilizado para utilizar como chave no *HashMap* para ser indexado no campo *campos*. Os valores que essa enumeração pode utilizar são "LESTE", "NORDESTE", "NORTE", "NOROESTE", "OESTE", "SUDOESTE", "SUL" e SUDESTE.

Terminado a descrição da enumeração explora a implementação do comportamento celular.

3.2.1 Comportamento Celular

O comportamento celular no software está implementado em um arquivo chamada *Cell.pde*, o qual descreve a classe *Cell*. Essa classe possui responsabilidade de definir o comportamento de cada célula que é instanciada no programa principal. Os comportamentos implementados são o movimento em direção a uma célula considerada alimento e o movimento da célula em uma

direção aleatória vizinha. O primeiro comportamento celular descrito nesse texto é a movimentação celular, ilustrada na Figura 3.16.

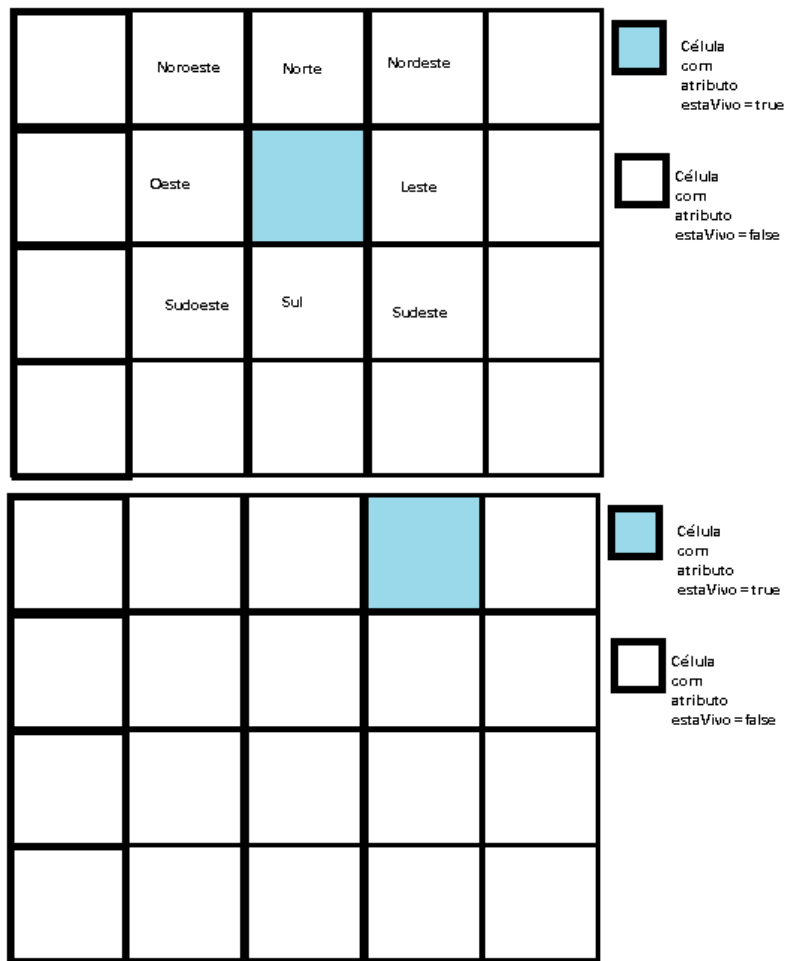


Figura 3.16: Exemplo de movimento celular para o vizinho Nordeste

A Figura 3.16 ilustra uma célula se movimentando para a direção Noroeste. O segundo comportamento celular implementado é o de farejar alimentos. A Figura 3.17 ilustra o comportamento de movimentar na direção de uma célula alimento.

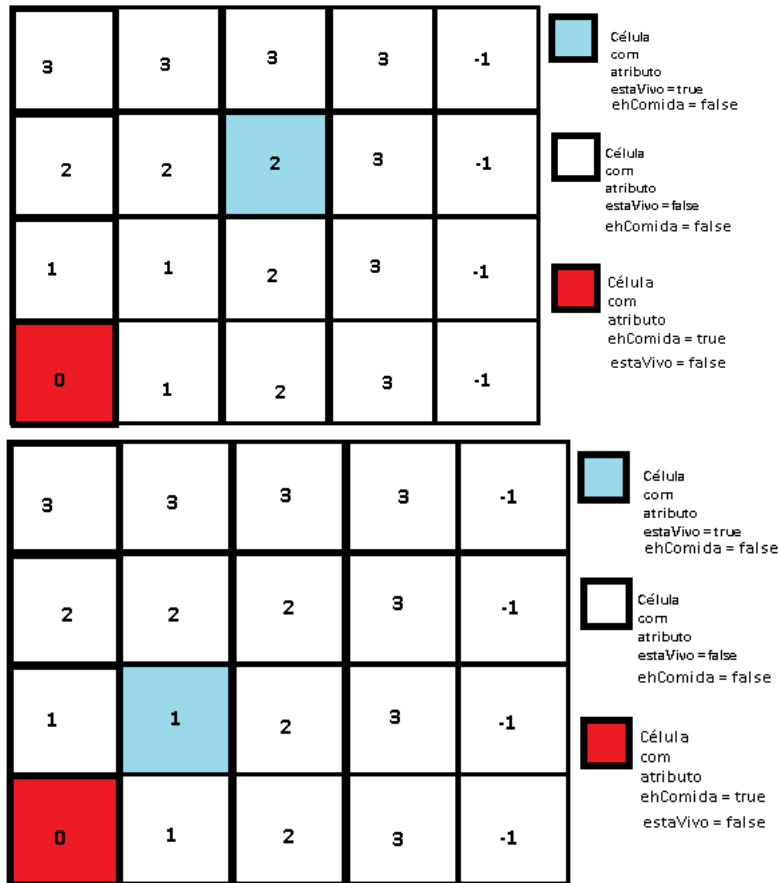


Figura 3.17: Exemplo de farejar alimento

A Figura 3.17 ilustra o comportamento de farejar alimentos. No caso a célula percebe que a posição a sudoeste dela possui um potencial menor em executar a mesma ação, portanto se move naquela direção. Eventualmente nesse contexto a célula se movimentará para uma posição de potencial 0. O raio de percepção é de 3 células de distância entre células vizinhas. Em outras palavras, a célula consegue perceber alimentos se as células alimentos serem vizinhos em uma distância de 3. Descrito o comportamento celular descrevemos a classe em detalhe como seus campos e métodos: A Classe *Cell* possui como campos descritos abaixo:

- Campos da classe:

- *Pvector* pos

Pvector é uma classe padrão do *Processing* de Vetor Euclidiano, que descreve a posição x, y, z de cada célula.

- *HashMap*<*Direcao*,*Cell*> neighbors

HashMap é uma classe de composição de chave e objeto, sendo a chave um valor da enumeração *Direcao* e o objeto do tipo *Cell*.

- *float* energy
o *float* energy é utilizado como uma forma de determinar tempo de vida da célula.
- *boolean* estaVivo
o *boolean* *estaVivo* é uma flag informando se a célula está viva ou não naquele instante.
- *boolean* ehFronteira
o *boolean* *ehFronteira* é uma flag informando se essa célula pode ser considerada como uma célula-fronteira e caso essa célula seja uma célula-fronteira impede outras células de se movimentar na *Direcao* indicada no hashMap.
- *boolean* frameTerminou
o *boolean* *frameTerminou* é uma flag informando se a célula já realizou alguma ação nesse frame, só irá ser permitido uma outra ação se a célula foi mostrada na tela.

Os métodos implementados na classe *Cell* são descritos abaixo:

- Métodos de construtor da classe

- *Cell(Pvector pos)*
A instanciação da classe inicia um *hashmap* de classe *Direcao* e *Cell*. Também são atribuídos aos campos de informação *frameTerminou*, *estaVivo*, *energy* são setados respectivamente *false*, *false* e 0.
- *Cell(boolean ehFronteira)*
As diferenças entre as duas assinatura o argumento *pos* é setado no campo *pos* e o outro método seta o campo *ehFronteira* como o argumento *ehFronteira*.

- Métodos de comportamento celular:

Uma célula é uma abstração de um *LED* na estrutura. As células podem estarem vivas, mortas, serem alimentos ou serem fronteiras do ambiente. Descreve-se os métodos que implementam o comportamento celular.

- *boolean* movimentaCelula(*Direcao* direcao)
O movimento celular em uma direção é implementado como um teste se o vizinho com a chave *direcao* é uma célula fronteira e se o frame terminou. A função *frameTerminou()* retorna um booleano verdade se o frame terminou, caso essa condição seja verdade o vizinho em questão recebe o valor *energy* da célula que está sendo movimentada, executa a função *terminaFrame()* para terminar o frame e não poder ocorrer mais nenhuma ação dessa célula.
- *void* movimentoRandomico()
O método randômico é implementado na função *movimentoRandomico()*, esse mé-

todo faz uma varredura com todos os vizinhos da célula corrente, a que está executando esse método, e caso a condição $random(5,0) < 0.5$ e $!this.neighbors(key).ehFronteira$, o que significa que a célula corrente não é uma célula fronteira e também o retorno da função *frameTerminou()*, o que significa que a célula já performou uma ação de se movimentar no valor *true*. Finalmente executa os métodos *fillUpEnergy()*, *celularDeath()* e *terminaFrame()* pela célula vizinha. O resultado disso é uma célula viva na *Direcao* direcao e a célula atual morre dando as propriedades para a célula viva nova.

– *void* farejaComida()

O comportamento da célula se movimentar na direção de uma célula alimento, *farejarComida()*, testa se a célula atual está possui algum valor positivo do campo *pathComida* que indica o valor do campo potencial em que a célula se encontra. Após isso é acessado os vizinhos através da lista do *hashMap* da célula corrente e caso exista um vizinho que não seja fronteira com um *pathComida* menor que o da célula corrente, ou seja uma célula com a condição *estaViva* como verdade, executa-se *movimentaCelula(direcao)*, sendo *direcao* a direção do vizinho com menor potencial, caso seja falsa a condição do potencial da célula corrente ser positivo ou não existir nenhum vizinho com potencial menor e não seja uma célula viva termina a célula corrente permanece inalterada.

– *void* becomeFoodCell()

O nascimento de uma célula que seja considerada alimento é implementado no método *becomeFoodCell()* que seta o *boolean ehcomida* e depois executa a função *atualizarCampoFood()*, que consiste de uma visita exaustiva de três(3) *for* aninhados varrendo todos os *hashmap* de todos os vizinhos da células, dando um raio de distância de três (3) células de distância. Caso um desses vizinhos não tenha um campo potencial ou o campo potencial que vai ser colocado é menor do que o que ele já possui e simultaneamente essas duas condições esse vizinho não ser uma célula fronteira é setado o potencial, *pathComida*, como o potencial da célula corrente na iteração da busca mais uma unidade.

– *void* expireFood()

As células que são consideradas alimentos são eliminadas com o passar do tempo , sendo necessário o usuário gerar mais movimento para criar mais células alimentos. Esse comportamento do alimento se deteriorar é obtido através do método *expireFood()*, que diminui a energia da célula de alimento em 25% e depois testa se a energia da célula é menor que o float 0.1, caso sim percorre-se os vizinhos da mesma forma do método *atualizarCampoFood()* e seta o potencial como o valor -1, isto é, campo potencial nulo, porém logo após é removido a célula de alimento da lista

de células de alimento e é executado o método `atualizarCampoFood()` em todas as células de alimento atuais de modo que reestabeleça o campo potencial das células alimentos no ambiente celular.

– `void atualizarCampoFood()`

Exemplifica-se o funcionamento do método `atualizarCampoFood()` em uma Figura, a Figura 3.18. As células são representadas por quadrados, as células vizinhas são representadas como quadrados adjacentes de outros quadrados, não necessariamente as células estão em estado vivo, isto é, com o valor *boolean estaVivo* como *true*. A Figura 3.18 mostra dois estados, um estado com apenas uma célula alimento, e outro estado logo após com duas células alimento.

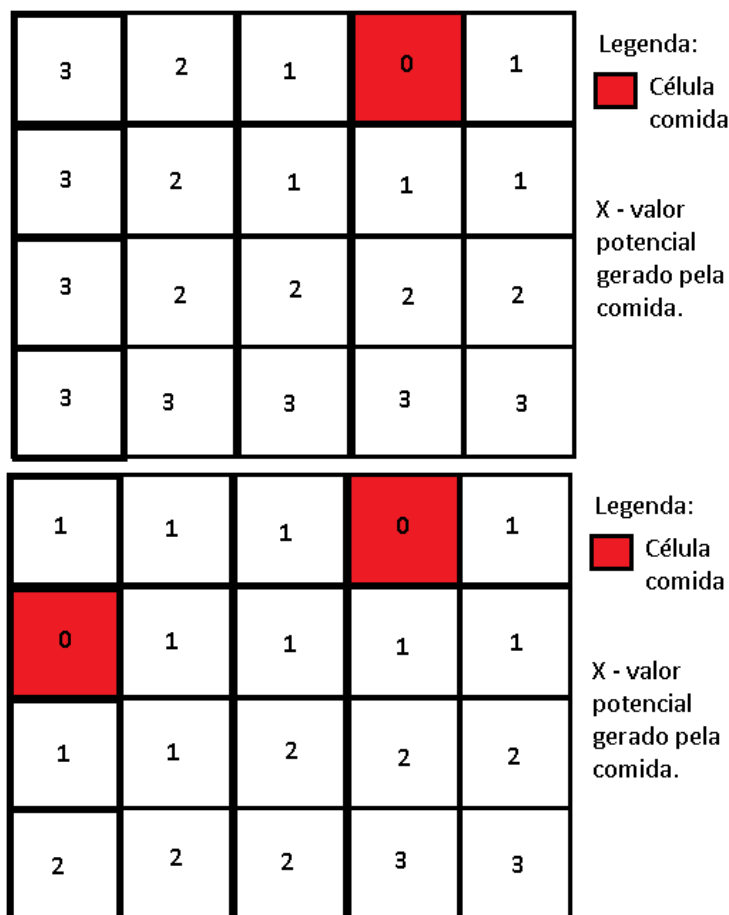


Figura 3.18: Estado T_0 , antes de um surgimento de uma nova alimento e o estado T_1 , depois do surgimento de uma nova alimento

A Figura 3.18 ilustra dois momentos, uma célula alimento e o outro momento com duas células alimento. Observa-se na Figura 3.18 que as células adjacentes e seu valor potencial são modificadas por as duas células alimento.

- *void* `addToNeighbors(Direcao direcao, Cell cell)`

O método `addToNeighbors()` implementa a adjacência de células. O método consiste de alimentar uma *HashMap* de referência para outras células, onde a chave é a Direção. Todo este funcionamento representa que células são vizinhas e que direção geograficamente estão. *boolean* `frameTerminou()`

O método `frameTerminou()` implementa setar o *boolean* `frameTerminou` como o valor lógico *false*. Esse *boolean* é utilizado para limitar os movimentos das células, cada célula só movimenta uma casa por frame.

3.2.2 Captura de Imagem

A responsabilidade de captura e estimativa de movimento é do arquivo chamado *Camera.pde*. O arquivo *Camera.pde* é o módulo de câmera. O módulo de câmera corresponde uma classe *Camera*, que é responsável por iniciar a câmera e utilizar os objetos de estimativa de cálculo. Esses objetos são descritos posteriormente na Subseções 3.2.4 e 3.2.4.2

A classe *camera* importa a biblioteca padrão do *Processing* chamada, *processing.video* [32].

O algoritmo de detecção de movimento na imagem capturada na biblioteca consiste de varredura pixel por pixel das duas imagens capturadas. Cada pixel será feito um ponto $P = (r, g, b)$ sendo que (r, g, b) é uma tripla de inteiros que indicam a codificação de cor em RGB. A classe *camera* possui uma série de métodos e campos. No contexto do programa o objeto *Camera* é apenas instanciado uma vez, esse objeto é responsável por estimar o movimento capturado pela câmera. Descreve-se na lista abaixo os campos da classe *Camera*:

- Campos da classe:

- *Capture* `cameraPrincipal`

Objeto abstração responsável por fazer as captações de frames vindas de uma webcam ligada a raspberry.

- *PImage* `frame1`

Objeto abstração que representa uma imagem básica no contexto do *Processing*, esse campos em particular representa o frame no instante t .

- *PImage* `frame2`

Objeto abstração que representa uma imagem básica no contexto do *Processing*, esse campos em particular representa o frame no instante $t + 1$.

- *float* `fluxo[][][]`

Um array de duas casas de matriz de float que representa o valor de uma das coordenadas de fluxo ótico em cada pixel da imagem da câmera, pode ser entendido também como uma matriz de uma dupla de floats.

– *float* fluxoAmbiente[][][]

Um array de duas casas de matriz de float que representa o valor de uma das coordenadas de fluxo ótico em cada pixel da imagem de um espaço de duas dimensões de tamanho arbitrário, pode ser entendido também como uma matriz de uma dupla de floats.

– *final int* LIMITE_MIN_X

o limite mínimo da coordenada horizontal de fluxo ótico, caso o valor seja menor que o valor dessa constante ele será desconsiderado.

– *final int* LIMITE_MIN_Y

o limite mínimo da coordenada vertical de fluxo ótico, caso o valor seja menor que o valor dessa constante ele será desconsiderado.

– *public OpenCvInterface* interfaceCV

O objeto abstração para utilizar as funcionalidades da biblioteca OpenCV no contexto desta classe.

– *Prototipo* ambiente

Abstração da aplicação do programa que é necessário para referenciar e desenhar na tela.

- Métodos construtor da classe

– cameraInput(Prototipo ambiente)

O construtor da classe funciona instanciando objetos *Capture*, *PImage* e *OpenCvInterface*, os campos *frame1* e *frame2* são atribuídos as *PImage*'s, o campo de *cameraPrincipal* é atribuído o objeto *Capture* e o campo *interfaceCV* é atribuído o objeto do *OpenCvInterface*. A instanciação é utilizada em um bloco de código *try* e *catch*, caso ocorra uma exceção o programa será terminado.

- Cálculo do fluxo ótico utilizando OpenCV e/ou Horn-Schunck

– *void* calculaFluxoOpenCv(*int* rowsAmbiente, *int* colsAmbiente)

O método *calculaFluxoOpenCv()* implementa a estimativa de fluxo ótico utilizando a biblioteca aberta *OpenCv*. Os argumentos são respectivamente a largura e altura em pixel do ambiente celular. O ambiente celular na obra tem largura cinco (5) e dez (10) células. Os campos *frame1* e *frame2* são filtrados para escalas de cinza através dos métodos da classe *PImage* e são usados no algoritmo de estimativa de fluxo ótico. Após execução do algoritmo de expansão polinomial a matriz de vetores está armazenada no campo *fluxoAmbiente*.

– *float[][][]* getFluxoAmbiente()

A função *getFluxoAmbiente()* é uma função que retorna uma matriz de vetores. Essa matriz de vetores está armazenada na matriz *uxoAmbiente*. Essa matriz está sempre armazenando o resultado dos algoritmos de estimativa de fluxo ótico. Porém como a imagem capturada é maior que o ambiente é necessário fazer uma aproximação entre a imagem e o ambiente celular. A aproximação utilizada é uma simples regra de 3.

3.2.3 Interface de Hardware

O segmento do projeto com responsabilidade de fazer a interface de hardware está no arquivo *InterfaceHardware.pde*. O arquivo *InterfaceHardware.pde* descreve uma classe com nome *HardwareInterface* que possui como responsabilidade de mandar sinais elétricos através dos pinos GPIO da *Raspberry* de forma que o display na obra seja correspondente a situação das células. A classe *HardwareInterface* possui uma série de métodos e campos os quais são descritos a seguir. Começa-se com os campos:

- Campos da classe:

- *boolean* dataOutput

- Ponteiro para um vetor de valores booleanos a serem passados para os periféricos do raspberry sequencialmente sendo o valor no índice 0 sendo o primeiro

- *boolean* limiteBits

- Número de bits a serem passados, também determina o tamanho do vetor a ser instanciado e apontado pelo *dataOutput*.

- Métodos construtor de classe:

- *HardwareInterface()*

- O método construtor da classe *HardwareInterface* é um método simples que instancia o array de booleanos para o tamanho adequado e o campo limite. O primeiro método assinatura tem como funcionamento o booleano com tamanho de 50.

- *HardwareInterface(int limite)*

- O método construtor da classe *HardwareInterface* é um método simples que instancia o array de *booleanos* para o tamanho adequado e o campo limite. O segundo método instancia com tamanho do parâmetro passado.

- Emitir um frame para os LEDs

– *Boolean* mandaTodaData()

Método que manda o sinal, através do pino GPIO4, para os periféricos do raspberry, o sinal que correspondente no esquemático do hardware é *SER*. São passados sinais de controle paralelamente, os sinais que correspondentes no esquemático do hardware é *SHIFT CLK e STORE CLK*, através do pino GPIO5 e GPIO6. A forma como os dados são mandados é baseado no funcionamento do circuito integrado 745HC595N.

Os sinais produzidos afim de representar a situação das células em cada determinado momento, são composto de três sinais paralelos. O primeiro dos três sinais paralelos foi utilizado para enviar sequências de bits. O display consiste de m colunas por n linhas de LEDs, portanto a sequência de bits para esse cenário seria de tamanho $m \times n$ bits, o pino correspondente na Tabela chama-se DS 2.4. O segundo sinal paralelo emite o sinal de clock de shift do shift register 745HC595, o pino correspondente na Tabela chama-se SHCP 2.4 O terceiro sinal paralelo emite o sinal de clock de store do shift register 745HC595, o pino correspondente na Tabela chama-se STCP 2.4

Terminada a explicação da implementação do código que faz a interface com *hardware* explora-se a implementação de fluxo ótico.

3.2.4 Implementações de estimativa de fluxo ótico

O módulo que possui como responsabilidade fazer estimativa de movimento dado uma sequência de frames, são implementados de duas formas diferentes. A primeira forma implementa o método expansão polinomial utilizando a biblioteca *OpenCV*. Explora-se essa implementação na Subseção 3.2.4.1. A segunda Subseção 3.2.4.2 explora a implementação proposta de *Horn & Schunck*.

3.2.4.1 Método de estimativa de expansão polinomial

A primeira forma está em um arquivo de nome *OpencvInterface.pde*. Esse arquivo possui uma classe com nome de *OpencvInterface* que possui como responsabilidade a biblioteca *OpenCV* do *Processing*. Essa biblioteca *OpenCV* faz a portabilidade da biblioteca *OpenCV* versão *Java*, atualmente na versão 2.4.5. Essa versão da biblioteca da *OpenCV* para *Processing* possui apenas um método de estimativa de fluxo ótico, a expansão polinomial. Para cumprir as responsabilidades atribuídas a essa classe foram implementados métodos para calcular o fluxo ótico, calcular fluxo ótico e enquadrar em um ambiente arbitrário. Primeiro descreve-se em detalhes o primeiro módulo, isto é, *OpencvInterface*. A classe *OpencvInterface* possui uma série de métodos e campos os quais são descritos a seguir. Começa-se com os campos:

- Campos da classe:

- *private int* rowsFrame

- O inteiro que indica no contexto da classe o número de linhas dos frames em que o processador da biblioteca OpenCV trabalhará.

- *private int* colsFrame

- O inteiro que indica no contexto da classe o número de colunas dos frames em que o processador da biblioteca OpenCV trabalhará.

- *private OpenCV* opencvProcessador

- O objeto abstração da biblioteca aberta OpenCV versão portátil do *Processing*, todo o processamento providenciado pela biblioteca é acessado através de um objeto instanciado OpenCV, no caso se utilizará o cálculo estimativo do fluxo ótico entre frames consecutivos.

- Estimativa do fluxo ótico em dois frames utilizando Opencv

- *float[][][][] calculaFarneback(PImage frame1, PImage frame2, int rowsAmbiente, colsAmbiente)*

- Esse método possui como responsabilidade utilizar a biblioteca do OpenCV para calcular o fluxo ótico pixel a pixel gerando um vetor como uma dupla de coordenadas sendo o eixo cartesiano como referencia para as coordenadas. A estimativa do fluxo ótico é calculado através uma média de 2 dimensões de 10x10 pixels utilizando a função *getTotalFlowInRegion()*. O método *getTotalFlowInRegion()* é uma das funções que utiliza o método expansão polinomial e está implementado na *Opencv*. Após retornar a estimativa do fluxo nessa região de 10x10 pixels, divide-se por 10^2 o módulo do fluxo estimado. Uma ultima correção é necessária pois o eixo y da coordenada está invertido, portanto se multiplica por -1 a coordenada vertical. O método também converte o fluxo ótico de uma resolução maior para uma menor, através de uma regra de três simples, $round((i*rowsAmbiente)/frame1.height)$ e $round((j*colsAmbiente)/frame1.width)$ respectivamente correspondem as linhas e as colunas da nova resolução, caso tenha dois vetores na mesma posição esses vetores são somados. Nos dois métodos é necessário inverter o eixo x também para se obter o efeito espelho em relação a uma pessoa em frente a obra como mostrado abaixo na Figura 3.1.

- Seja o frame de tamanho $M \times N$ e a resolução passada no segundo método $L \times K$, os métodos retornam respectivamente $float[M][N][2]$ e $float[L][K][2]$, sendo o eixo de tamanho 2 devido a cada posição na matriz possuir uma dupla de coordenadas, $\vec{h} = (u, v)$. Após ter descrito como funciona o módulo OpenCV descreve-se a próxima

Subseção que implementa o mesmo cálculo estimativo do fluxo ótico utilizando o método *Horn-schunck*.

Terminada a explicação da implementação da estimativa pelo método expansão polinomial inicia-se a explicação da implementação pelo método *Horn-Schunck*.

3.2.4.2 Módulo de estimativa Horn-Schunck

O segundo módulo que faz a estimativa de fluxo ótico se chama *Horn-Schunck.pde*. O módulo possui uma classe chamada *Fluxo* que utiliza objetos do tipo *convolucaoOperador*. Os objetos *convolucaoOperador* auxiliam fazendo operações necessárias para a estimativa de fluxo ótico no método *Horn & Schunck*. Descreve-se agora a classe que tem como responsabilidade executar o algoritmo. Dentro do contexto do programa só é usado uma instância para desempenhar essa tarefa.

- Campos da classe:

Inicia a descrição da classe começando pelos campos. Logo após é descrito o funcionamento através dos métodos e funções.

- *private float[][][]* fluxo

Essa matriz armazena globalmente o fluxo compartilhado por todas as classes. Esse fluxo é iniciado com 0 em todas os seus pares ordenados. O fluxo atual é representado no conteúdo dessa matriz.

- *private Threads t1,t2,t3* Objetos da classe

Threads que implementam a execução paralela. Cada *Thread* inicia uma das classes filhas da classe *convolucaoOperador*. Respectivamente cada um calcula separadamente o resultado das derivadas de duas dimensões. Essas derivações são as referenciadas na equações 2.15, 2.16 e 2.17.

- *private float* LIMITE_MIN_X

Constante de valor minimo para ser considerado um vetor de movimento na coordenada horizontal.

- *private float* LIMITE_MIN_Y

Constante de valor minimo para ser considerado um vetor de movimento na coordenada vertical.

- *private float* frameXFloat[][]

Matriz de inteiros onde será armazenado a derivada de duas dimensões da coordenada horizontal do primeiro frame.

- *private float* frameYFloat[][]
Matriz de inteiros onde será armazenado a derivada de duas dimensões da coordenada vertical do primeiro frame.
- *private float* frame1T[][]
Matriz de inteiros onde será armazenado a derivada de duas dimensões da coordenada temporal do primeiro frame.
- *private int* kMax
A constante que será utilizada como a condição de parada
- *private float* lambda
A constante de suavidade do método Horn-Schunck e λ .
- *private convolucaoOperador* operador
O objeto de que possui como responsabilidade de cálculo de convoluções, isto é, as derivadas parciais em coordenadas eixo horizontal, vertical e temporal.

- Estimar o fluxo a partir de dois frames

- *float* [][][] estimarFluxo(int frame1[][], int frame2[][])
A implementação já explicada direta e *naive*, utilizando o objeto do tipo *convolucaoOperador* para cálculos das derivadas parciais das coordenada horizontais, verticais e temporais, o cálculo é de acordo com as equações 2.11 e 2.12 pixel à pixel e são somados a diferença entre o valor do pixel da ultima iteração elevada ao quadrado, de acordo com a equação 2.21, após o cálculo estimativo retorna uma instância de um array de matriz de vetores de *float*. O funcionamento desse cálculo necessita operações de convoluções de matrizes, no caso matrizes de pixels e matrizes *kernel*. Foi implementado uma classe que calcula essas convoluções de matrizes, utiliza-se o objeto no contexto do programa. Essa classe implementada se chama *ConvolucaoOperador*.

Terminada a explicação do módulo principal que implementa o *Horn & Schunck*, inicia a explicação da classe *convolucaoOperador.pde*. Esta classe implementa o método *run* necessário para se usar *Threads* do Java. As *Threads* são linhas de execução paralela. Foi criado três classes filhas que estendem essa classe *convolucaoOperador* e implementam o próprio método *run*. As classes filhas são chamadas de *derivarHorizontal*, *derivarVertical* e *derivarTemporal*. Respectivamente os métodos *run* dessas classes executam os resultados dos métodos *derivarXOtimizado*, *derivarYOtimizado* e *derivarT1T2Otimizado*. Descreve-se logo a seguir os campos que todos esse métodos filhos compartilham em comum:

- Campos da classe:

- *private int* `derivateXOtimizadoCol[]`
Um array de inteiros que representa a matriz kernel de derivação em relação ao eixo x, separada em um produto de matrizes linha e coluna, é inicializado com o valor do array -1,-1.
- *private int* `derivateXOtimizadoLin[]`
Um array de inteiros que representa a matriz kernel de derivação em relação ao eixo x, separada em um produto de matrizes linha e coluna, é inicializado com o valor do array 1,-1.
- *private int* `derivateYOtimizadoCol[]`
Um array de inteiros que representa a matriz kernel de derivação em relação ao eixo y, separada em um produto de matrizes linha e coluna, é inicializado com o valor do array 1,-1.
- *private int* `derivateYOtimizadoLin[]`
Um array de inteiros que representa a matriz kernel de derivação em relação ao eixo y, separada em um produto de matrizes linha e coluna, é inicializado com o valor do array 1,1.
- *private int* `derivateOtimizadoT1Col[]`
Um array de inteiros que representa a matriz kernel de derivação em relação ao eixo temporal, separada em um produto de matrizes linha e coluna, é inicializado com o valor do array -1,-1, para ser usado o momento posterior é necessário multiplicar por -1, para ser usado os valores positivos invés dos negativos.
- *private int* `derivateOtimizadoT1Lin[]`
Um array de inteiros que representa a matriz kernel de derivação em relação ao eixo temporal, separada em um produto de matrizes linha e coluna, é inicializado com o valor do array 1,1.
- *float* `media[][]`
A matriz kernel gaussiana necessária para fazer o cálculo da vizinhança via convolução de duas dimensões.
- *private int* `kCols`
Número de colunas nos kernels de convolução.
- *private int* `kRows`
Número de linhas nos kernels de convolução.
- *private int* `mediaCols`
Número de colunas no kernel gaussiana de convolução.
- *private int* `kRows`
Número de linhas no kernel gaussiana de convolução.

Os cálculos de derivação matricial são implementadas nos métodos listados abaixo:

- Derivações Matriciais

- *int*[][] derivarXOtimizado(*int* in1[], *int* in2[])

- *int*[][] derivarYOtimizado(*int* in1[], *int* in2[])

- *int*[][] derivarT1T2Otimizado(*int* in1[]M[], *int* in2[])

A forma implementada para todas as derivações de duas dimensões em relação a coordenada horizontal, vertical e temporal de uma imagem. Os argumentos passados são frames seguidos o que é necessário para o cálculo numérico das derivadas. Utiliza-se a convolução separável com os kernels de derivação horizontal, vertical e temporal. As convoluções que são feitas nessa classe são as indicadas nas equações 2.15, 2.16 e 2.17 para derivação horizontal, vertical e temporal respectivamente. O final das funções é o retorno da matriz de inteiros que contém o resultado do cálculo. Após depois de explorar o funcionamento do cálculo derivacional, explora-se o cálculo de média da vizinhança em duas dimensões.

- Cálculo da média da vizinhança da matriz

- *float*[][][] calcularMediaVetor(*float* in1[][][])

O cálculo da média da vizinhança de pixels necessário para o cálculo laplaciano, simultaneamente o fluxo ótico nas duas coordenadas, o argumento in1 vai ser de onde vai ser processado a média, a média a ser usada será a gaussiana, equação 2.19. Retorna-se uma instância de uma dupla de matriz de *float* com a média da vizinhança processada pixel a pixel. Terminada a explicação do funcionamento da média gaussiana começa o texto que explicará o classe Horn-Schunck.

Terminada a explicação de como foi o processo de produção a próxima Seção contém o processo de fabricação da carcaça de madeira que funciona como estrutura da obra.

3.3 Manufatura Carcaça

Nesta Seção contém o processo a ser seguido afim de obter uma construção de madeira que irá servir de estrutura para a obra. A fabricação da estrutura de madeira foi feito por um serviço terceirizado. O serviço terceirizado foi criado na primeira versão do projeto *Aura Vitallis* [8]. Terminada a escrita do processo anterior de construção da estrutura de madeira, descreve-se a maneira que foi executada a apresentação da obra ao público.

Capítulo 4

Resultados e Análise

Este capítulo descreve os resultados seguindo os passos descritos na seção anterior. Comentários, discussões e ponderações relativas aos resultados estão inclusas. A primeira Seção 4.1 será a apresentação do desempenho do algoritmo de visão computacional. A segunda Seção 4.2 explora a apresentação da obra em um espaço de arte contemporânea.

4.1 Resultados

Esta seção descreve o resultado da interação usando visão computacional. Os resultados podem ser separados em duas subseções. A Subseção 4.1.1 descreve o tempo de processamento obtido pelo software e qual o seu impacto. A segunda Subseção 4.1.2 descreve a verificação e validação do programa utilizando mensurações do instituto *Middlebury*.

4.1.1 Tempo de processamento

Esta subseção apresenta os dados desempenho do programa em três *datasets* para ilustrar o comportamento do programa. O desempenho descrito nessa subseção é o tempo de processamento. A Subseção 4.1.1.1 descreve o estudo utilizando o benchmark *Middlebury* [4]. A Subseção 4.1.1.2 descreve o estudo utilizando o benchmark *Kitti* [5]. A Subseção 4.1.1.3 estuda utilizando um dataset próprio feito pensando na interação da obra. A última subseção explora a experiência que a interação planeja resultou e suas limitações e dificuldades.

4.1.1.1 Benchmark *Middlebury*

O tempo de processamento é utilizado para realizar a comparação das duas implementações do cálculo do fluxo ótico. Obtém-se uma sequência de tempos de processamento, cada par de *frames* foram processados uma série de cinquenta (50) vezes seguidas. A sequência de *frames* são utilizadas de um *dataset* focado em construir *benchmarks* de fluxo ótico de Avaliação [4].

O *dataset* middlebury consiste da sequência de par de imagens que estão no anexo III [4]. Um exemplo de imagens utilizadas no *benchmark* estão ilustrados pela Figura 4.1 [4].



Figura 4.1: Frames seguidos de exemplos do dataset [4]

Após seguido os procedimentos descritos anteriormente obteve-se o resultado de tempo de processamento por *frames*. Esse *benchmark* gerado é processado no hardware da *Raspberry PI* modelo 2B. Os parâmetros do algoritmo expansão polinomial é descrito na lista abaixo, caso o parâmetro não seja descrito ele é o padrão da biblioteca:

- Pyramid scale = 0.5
- $\sigma = 1.0$

Os parâmetros do algoritmo *Horn & Schunck* são descritos na lista abaixo:

- $\alpha = 15$
- $\varepsilon = 0.5$

Os resultados de tempo processamento são explicitados em um gráfico de *número da iteração do processamento por tempo de processamento*, o eixo horizontal do gráfico indica qual iteração e o eixo vertical indica o tempo de processamento daquela interação. As séries que estão no gráfico são etiquetadas com o nome da imagem que estão no anexo, isto é a série de imagens a Figura III.1 até a Figura III.12, e indica qual era a imagem sendo processada na hora, as cores as distinguem. Toda essa informação é apresentada nos gráficos na Figura 4.3.

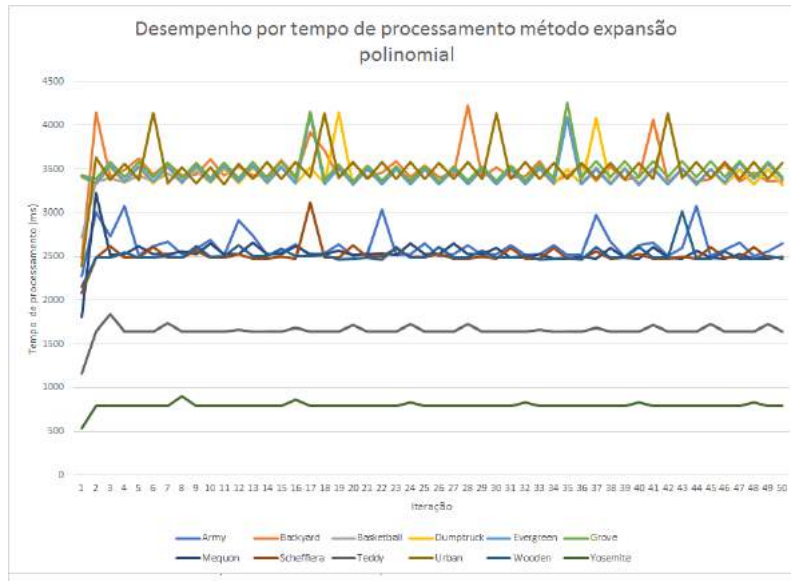


Figura 4.2: Gráfico de número do teste por tempo de processamento *dataset Middlebury*, método expansão polinomial.

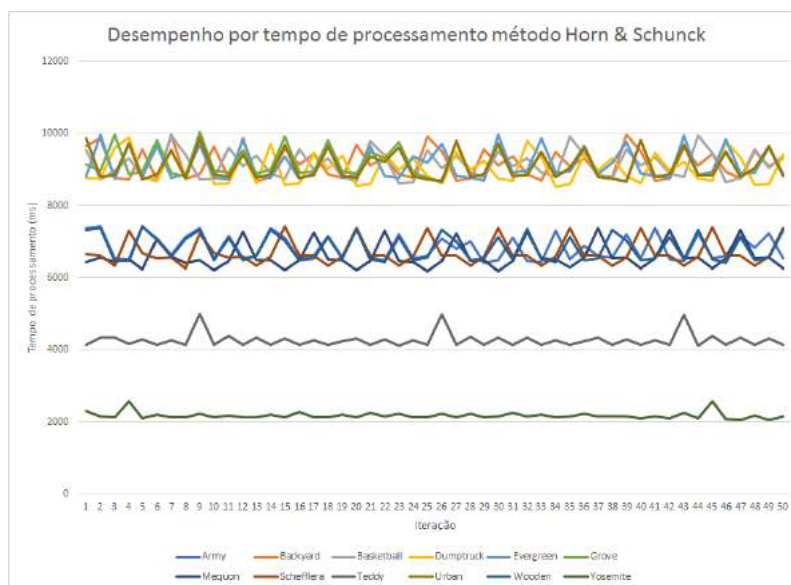


Figura 4.3: Gráfico de número do teste por tempo de processamento *dataset Middlebury*, método Horn & Schunck.

A partir das Figuras 4.3 e 4.2 observa-se uma variação de desempenho entre processamentos consecutivos nos dois métodos, ambos métodos necessitam um tempo variável para processar dois *frames*. O método *Horn & Schunck* apresenta uma variação maior de tempo de processamento entre sucessivos processamentos. Outro aspecto observado é o tempo de resposta da implementação do *Horn & Schunck* é muito maior que a implementação da Expansão Polinomial. Gera-se a Tabela 4.1 a partir do gráfico na Figura 4.3. A Tabela 4.1 apresenta, para os

dois métodos, o tempo médio de processamento entre todas iterações do gráfico das Figuras 4.3 e 4.2. Além disso a tabela 4.1 apresenta o tempo de processamento médio de cada método sobre cada imagem e o fator de diferença entre eles. O fator de diferença é a divisão do tempo do método *Horn & Schunck* dividido pelo tempo do método expansão polinomial.

Tabela 4.1: Tempo de processamento médio em milisegundos pelos métodos expansão polinomial e Horn Schunck, *dataset Middlebury* de Avaliação , Quad-Core ARM cortex-A7 900Mhz e 1 GB RAM.

Imagem	Expansão Polinomial	Horn & Schunck	Fator diferença
Army(Hidden)	2623,94	6846,22	2,609
Backyard(HighSpeed)	3496,02	9183,36	2,627
Basketball(HighSpeed)	3430,46	9148,14	2,667
Dumptruck(HighSpeed)	3442,22	9055,48	2,630
Evergreen(HighSpeed)	3451,5	9153,78	2,652
Grove (Synthetic)	3505,08	9161,64	2,614
Mequon(Hidden)	2533,36	6585,06	2,600
Schefflera(Hidden)	2514,02	6696,30	2,664
Teddy	1653,24	4267,80	2,582
Urban(Synthetic)	3505,52	9111,22	2,599
Wooden(Hidden)	2512,42	6830,06	2,719
Yosemite(Synthetic)	798,18	2175,82	2,726

Analisando a Figura 4.3 podemos confirmar novamente o que os estudos de performance exploram, o método de expansão polinomial é mais veloz em seu processamento. A diferença entre o processamento do método Horn & Schunck e o método da expansão polinomial. Outra análise feita é a constância do fator de diferença independente das imagens e suas características. Muda-se o Hardware para um Hardware mais capaz, utiliza-se o mesmo *dataset* também. O hardware utilizado é um processador core i7 3635 com frequência 2,4 GHz e memória RAM de 8 GB. A tabela 4.2 mostra os resultados do tempo de resposta.

Tabela 4.2: Tempo de processamento médio em milissegundos pelos métodos expansão polinomial e Horn & Schunck, dataset Middlebury, processador core i7 -3635 @ 2,4 GHz e RAM 8 GB

Imagem	Expansão Polinomial(ms)	Horn & Schunck(ms)	Fator de comparação
Army(Hidden)	89,26	154,38	1,729
Backyard(HighSpeed)	120,10	204,02	1,699
Basketball(HighSpeed)	121,26	207,3	1,709
Dumptruck(HighSpeed)	121,38	203,76	1,679
Evergreen(HighSpeed)	120,64	202,70	1,680
Grove(Synthetic)	118,02	203,34	1,723
Mequon(Hidden)	87,82	149,10	1,698
Schefflera(Hidden)	88,14	148,62	1,686
Teddy(Stereo)	60,36	99,72	1,652
Urban(Synthetic)	119,04	202,82	1,704
Wooden(Hidden)	90,70	152,8	1,685
Yosemite(Synthetic)	29,00	50,64	1,746

A Tabela 4.2 possui o tempo em milissegundos dos dois métodos e o fator de diferença. O fator de diferença é calculado pelo quociente do tempo de processamento do *Horn & Schunck* pelo *expansão polinomial*. Portanto nesse *dataset KITTI* conclui-se que o método expansão polinomial foi mais veloz que o método de *Horn & Schunck*. Tendo em vista a conclusão anterior que o fator de diferença foi um valor constante de 2.7, compara-se com o fator de diferença da Tabela 4.2. O fator da tabela 4.2 flutua no valor 1.7 aproximadamente. Portanto a estimativa de fluxo ótico possui uma diferença de desempenho menor do que no hardware da *Raspberry pi*. Essa diferença de fator ocorre pela diferença de hardware entre os dois testes, o *Raspberry Pi* possui 900MHz comparado com processador core i7 -3635 com 2,4 GHz. A diferença das duas máquina também afeta, o *Raspberry Pi* possui sete GB a menos de memória RAM.

Uma outra implementação é testada, essa implementação utilizando os *Threads* para fazer as derivações matriciais. Utiliza-se o mesmo algoritmo *Horn & Schunck* porém apenas com a diferença da paralelização. Utiliza-se três *Threads* para derivação matricial na coordenada horizontal, vertical e temporal. Portanto em um processador que possui capacidade de execução em paralelo o ganho de tempo de resposta diminui. Apresenta-se o benchmark utilizando o *dataset Middlebury* novamente. O hardware que executa esse benchmark é a *Raspberry Pi*, *Quad-core ARM cortex-A7 900MHz* e 1GB de memória. A Tabela 4.3.

Tabela 4.3: Tempo de processamento médio em milissegundos pelos métodos expansão polinomial e Horn & Schunck utilizando *Threads*, *dataset Middlebury*, Quad-Core ARM cortex-A7 900Mhz e 1 GB RAM

Imagem	Expansão Polinomial	Horn & Schunck Paralelo	Fator Diferença
Army(Hidden)	89	34	0,38
Backyard(HighSpeed)	119,5	45	0,38
Basketball(HighSpeed)	120	45	0,37
Dumptruck(HighSpeed)	121	45	0,37
Evergreen(HighSpeed)	119	45	0,38
Grove(Synthetic)	119	45	0,38
Mequon(Hidden)	86	33	0,38
Schefflera(Hidden)	87	33	0,38
Teddy(Stereo)	58	22	0,38
Urban(Synthetic)	119	45	0,38
Wooden(Hidden)	87	33	0,38
Yosemite(Synthetic)	29	12	0,41

A Tabela 4.3 descreve os resultados do teste com a implementação *Horn & Schunck* em paralelo e a implementação de expansão polinomial. Os tempos da tabela são mostrados em milissegundos. O fator de diferença é o quociente entre o tempo *Horn & Schunck* dividido pelo tempo da expansão polinomial. A Tabela 4.4 compara as duas implementações do *Horn & Schunck*.

Tabela 4.4: Tempo de processamento médio em milissegundos pelos métodos Horn & Schunck Serial e Horn & Schunck Paralelo utilizando *Threads*, dataset Middlebury, Quad-Core ARM cortex-A7 ARM cortex-A7 900MHz e 1 GB RAM

Imagem	Horn & Schunck Serial	Horn & Schunck Paralelo	Fator Diferença
Army(Hidden)	154,38	34	0,22
Backyard(Highspeed)	204,02	45	0,22
Basketball(Highspeed)	203,76	45	0,22
Dumptruck(Highspeed)	202,7	45	0,22
Evergreen(Highspeed)	203,34	45	0,22
Grove(Synthetic)	149,1	45	0,30
Mequon(Hidden)	148,62	33	0,22
Schefflera(Hidden)	99,72	33	0,33
Teddy(Stereo)	202,82	22	0,10
Urban(Synthetic)	152,8	45	0,29
Wooden(Hidden)	50,64	33	0,66
Yosemite(Synthetic)	29	12	0,41

A Tabela 4.4 descreve os resultados do teste com a implementação *Horn & Schunck* em paralelo e a implementação não em paralelo. Os tempos da tabela estão em milissegundos. O fator de diferença é o quociente entre o tempo *Horn & Schunck* paralelizada dividido pelo tempo *Horn & Schunck* não paralelizado (serial). Analisando as Tabelas 4.3 e 4.4 observamos o comportamento das três implementações. A implementação *Horn & Schunck* paralelizada possui um fator de diferença 0,38 em relação ao tempo da expansão polinomial no benchmark 4.3. A implementação *Horn & Schunck* paralelizada possui um fator de diferença 0,30 em relação ao tempo da *Horn & Schunck* não paralelizada no benchmark 4.4. A implementação proposta nesse trabalho do método *Horn & Schunck* obteve um tempo de resposta menor. O ganho em relação a antiga implementação é aproximadamente 60%. Em relação a implementação utilizando *OpenCV* do método expansão polinomial teve um ganho de aproximadamente 61%. O ganho de desempenho entre as duas implementações do método *Horn & Schunck* é intuitivo, a paralelização de um processamento diminui o tempo de processamento. Porém o resultado entre as implementações *Horn & Schunck* e *expansão polinomial* é notável. O trabalho feito na biblioteca *Opencv* é extensivo e de qualidade, porém além da estimativa de fluxo ótico existe outro processamento pirâmide. O pós processamento de pirâmide é necessário para grandes deslocamentos na imagem porém tendo o custo em tempo de processamento. Terminado o estudo de tempo de resposta utilizando o *dataset Middlebury* descreve-se o estudo feito utilizando *KITTI*.

4.1.1.2 Benchmark *KITTI*

Utiliza-se também o *dataset* pertencente ao instituto *KITTI* de *benchmarks* de visão computacional. O *dataset* consiste de um banco de imagens utilizado com *benchmark* por vários autores [5]. O processamento ocorre da mesma maneira na seção 4.1.1.1. O processamento das imagens do *dataset* são executadas 50 vezes seguidas. O Hardware que executa o teste é a *Raspberry Pi* Modelo 2B com 900MHz. Um exemplo do banco de imagens está na Figura 4.4, abaixo [5], o *dataset* em sua totalidade está no anexo II.



Figura 4.4: Imagem número 00 frame 11

A relação de fotos utilizadas nessa análise de desempenho estão no anexo, a série de imagens II.1 até II.20. Os resultados a serem comparados são implementações dos mesmos dois métodos, expansão polinomial e *Horn & Schunck*. Os parâmetros do algoritmo expansão polinomial é descrito na lista abaixo, caso o parâmetro não seja descrito ele é o padrão da biblioteca:

- Pyramid scale = 0.5
- $\sigma = 1.0$

Os parâmetros do algoritmo *Horn & Schunck* são descritos na lista abaixo:

- $\alpha = 15$
- $\varepsilon = 0.5$

Utiliza-se do mesmo método para medir desempenho sobre outro banco de imagens, o *dataset* do instituto *KITTI*, o resultado do processamento das 20 imagens utilizadas no *dataset* são demonstradas na Figura 4.6 que contém gráfico de número do teste por tempo de processamento. A partir do teste de *benchmark* do *dataset KITTI* calcula-se a média de processamento em 50 execuções consecutivas de um mesmo par de imagem e a Tabela 4.5.

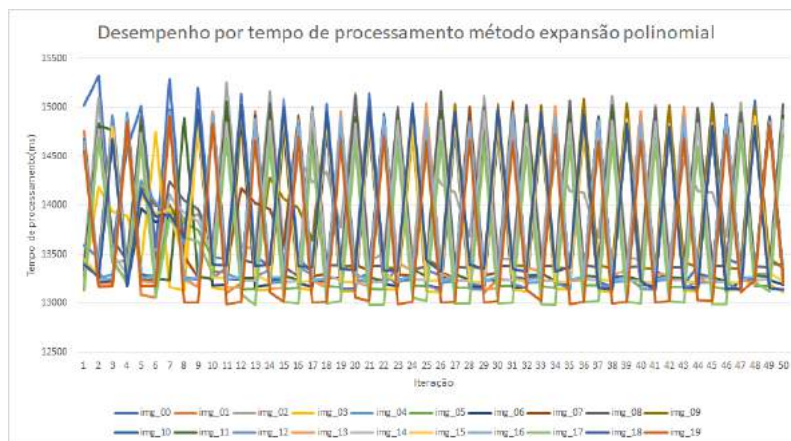


Figura 4.5: Gráfico de número do teste por tempo de processamento *dataset KITTI*, método Horn & Schunck.

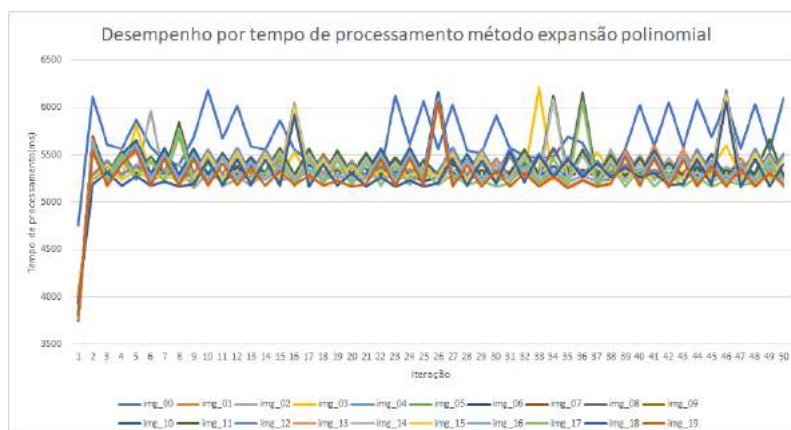


Figura 4.6: Gráfico de número do teste por tempo de processamento *dataset KITTI*, método expansão polinomial.

A partir das Figuras 4.5e 4.6 observa-se variação de desempenho entre processamentos consecutivos novamente, como foi observado no estudo na Subseção 4.1.1.1. Novamente o tempo de resposta da implementação do *Horn & Schunck* é muito maior que a implementação da Expansão Polinomial. Gera-se também a partir da Figura 4.6 a tabela abaixo 4.5. Na Tabela 4.5 também está incluso o fator de diferença que é o quociente entre o tempo *Horn & Schunck* e o tempo expansão polinomial.

Tabela 4.5: Tempo de processamento médio em millisegundos pelos métodos expansão polinomial e Horn & Schunck, *dataset KITTI*.

Imagem	Expansão polinomial (ms)	Horn & Schunck (ms)	Fator de Diferença
0	5660,22	13966,58	2,47
1	5371,38	13780,10	2,57
2	5331,24	14091,62	2,64
3	5352,48	13688,52	2,55
4	5316,02	13788,4	2,60
5	5313,08	13746,76	2,59
6	5328,92	13724,12	2,57
7	5319,02	13852,76	2,60
8	5356,58	13951,58	2,60
9	5369,30	13930,20	2,59
10	5368,50	13813,02	2,58
11	5365,94	13850,86	2,58
12	5372,64	13777,64	2,56
13	5301,22	13795,76	2,60
14	5317,72	13809,00	2,60
15	5328,94	13778,34	2,59
16	5319,16	13808,06	2,60
17	5275,52	13601,94	2,59
18	5281,12	13828,02	2,62
19	5262,16	13619,72	2,59

Utilizando-se das informações da tabela 4.5 verifica-se novamente a diferença entre as implementações e métodos. O Fator de diferença manteve-se constante entre os *dataset Middlebury* e *KITTI*. O valor de diferença manteve-se aproximadamente em 2.6. Novamente utiliza-se o mesmo *dataset KITTI* com o mesmo hardware utilizado na tabela 4.2.

Tabela 4.6: Tempo de processamento médio em milissegundos pelos métodos expansão polinomial e Horn & Schunck, dataset KITTI, Hardware core i7 -3635 @ 2,4 GHz

Imagem	Expansão Polinomial(ms)	Horn & Schunck(ms)	Fator de comparação
0	184,6	315,6	1,709
1	180,52	308,02	1,706
2	181,1	306,58	1,692
3	181,5	304,58	1,678
4	179,32	306,12	1,707
5	181,06	307,56	1,700
6	189,92	315,42	1,660
7	179,32	305,94	1,706
8	180,78	308,4	1,705
9	181,84	309,62	1,703
10	180,7	308,4	1,707
11	183,02	307,72	1,681
12	180,62	307,36	1,702
13	178	306,08	1,720
14	183,32	309,02	1,690
15	186,2	306,98	1,649
16	180	305,3	1,696
17	179,54	303,8	1,692
18	179,84	305,26	1,697
19	181,24	304,26	1,679

A Tabela 4.6 apresenta o tempo em milissegundos dos dois métodos e o fator de diferença. O fator de diferença é calculado pelo quociente do tempo de processamento do *Horn & Schunck* pelo *expansão polinomial*. Tendo em vista a conclusão anterior que o fator de diferença foi um valor constante de 2.6, compara-se com o fator de diferença da Tabela 4.6. O fator da Tabela 4.6 flutua no valor 1.6 aproximadamente. Portanto a estimativa de fluxo ótico possui uma diferença de desempenho menor do que no hardware da *Raspberry pi*.

Tabela 4.7: Tempo de processamento médio em milissegundos pelos métodos expansão polinomial e Horn & Schunck paralelizado, *dataset KITTI*.

Imagem	Expansão Polinomial	Horn & Schunck	Fator Diferença
img00	5480	8215	1,49
img01	5367	8029	1,49
img02	5342	8087,5	1,51
img03	5315	8008,5	1,50
img04	5254,5	8046,5	1,53
img05	5353	7976	1,49
img06	5332	8080	1,51
img07	5272	7949,5	1,50
img08	5304,5	7979,5	1,50
img09	5294	8080	1,52
img10	5258,5	8069	1,53
img11	5327,5	8079	1,51
img12	5426	7975	1,46
img13	5215,5	8071,5	1,54
img14	5201	8089,5	1,55
img15	5270	8268	1,56
img16	5201	7979	1,56
img17	5200,5	7960	1,53
img18	5198	8076	1,55
img19	5261	8013	1,52

A Tabela 4.7 descreve os resultados do teste com a implementação *Horn & Schunck* em paralelo e a implementação de expansão polinomial. Os tempos da tabela são em milissegundos. O fator de diferença é o quociente entre o tempo *Horn & Schunck* dividido pelo tempo da expansão polinomial. Utiliza-se a Tabela 4.7 para análise. Continua-se as comparações do método *Horn & Schunck*. Compara-se as duas implementações do método na Tabela 4.8.

Tabela 4.8: Tempo de processamento médio em milissegundos pelos métodos *Horn & Schunck* paralelizado e *Horn & Schunck* não paralelizado (serial), *dataset KITTI*.

Imagem	Horn Schunck Serial	Horn & Schunck Paralelo	Fator Diferença
img00	13966,58	8215	0,58
img01	13780,10	8029	0,58
img02	14091,62	8087,5	0,57
img03	13688,52	8008,5	0,58
img04	13788,4	8046,5	0,58
img05	13746,76	7976	0,58
img06	13724,12	8080	0,58
img07	13852,76	7949,5	0,57
img08	13951,58	7979,5	0,57
img09	13930,20	8080	0,58
img10	13813,02	8069	0,58
img11	13850,86	8079	0,58
img12	13777,64	7975	0,57
img13	13795,76	8071,5	0,58
img14	13809,00	8089,5	0,58
img15	13778,34	8268	0,60
img16	13808,06	7979	0,57
img17	13601,94	7960	0,58
img18	13828,02	8076	0,58
img19	13619,72	8013	0,58

A Tabela 4.8 descreve os resultados do teste com a implementação *Horn & Schunck* em paralelo e a implementação não paralela. Os tempos da tabela são em milissegundos. O fator de diferença é o quociente entre o tempo *Horn & Schunck* paralelizada dividido pelo tempo *Horn & Schunck* não paralelizado (serial). Analise-se as Tabelas 4.7, 4.8. A implementação *Horn & Schunck* não paralelizada possui um fator de diferença 1,53 em relação ao tempo da expansão polinomial no benchmark 4.7. A implementação *Horn & Schunck* não paralelizada possui um fator de diferença 0,58 em relação ao tempo da *Horn & Schunck* não paralelizado no benchmark 4.8. Observa-se que o tempo de resposta do método *Horn & Schunck* na implementação paralelizada não conseguiu ser mais veloz que todo o processamento da biblioteca *Opencv*. O método *Horn & Schunck* teve que ter mais iterações para o cálculo do fluxo. O método *Horn & Schunck* necessita de um número maior de iterações quando ocorre grandes deslocamentos. O *dataset* Kitti dispõe de imagens de resolução relativamente grandes e com deslocamentos grandes. Portanto mesmo a implementação *Horn & Schunck* paralela não pos-

suir pós-processamento ela gasta mais tempo de processamento. Portanto a implementação com expansão polinomial utilizando *Opencv* é mais indicado para imagens com grandes deslocamentos. Dado essas observações confirma-se que entre as duas implementações de estimativa de fluxo ótica o método de expansão polinomial é mais adequada para a *Raspberry Pi* no contexto de tempo real devida pouca capacidade computacional apresentada. A implementação da expansão polinomial é mais adequada para uma resposta em tempo real a qual a obra necessita. Terminada a análise de tempo de resposta utilizando o *dataset Kitti* inicia-se o estudo utilizando *dataset* próprio para interação da obra.

4.1.1.3 Benchmark dataset interação

Após ter sido feito o estudo utilizando o *dataset* do instituto *KITTI* é estudada utilizando outro *dataset*. No caso dessa Subseção utiliza-se o *dataset* indicado no anexo IV. Esse *dataset* foi criado com imagens que são as mais próximas pensadas para o funcionamento real da obra. O público interage através de detecção de movimento em imagens com a mesma resolução. A imagem 4.7 apresentada é um exemplo.



Figura 4.7: Imagem *dataset* interação.

Os resultados dos testes são demonstrados através da Tabela 4.9. Os parâmetros do algoritmo expansão polinomial é descrito na lista abaixo, caso o parâmetro não seja descrito ele é o padrão da biblioteca:

- Pyramid scale = 0.5
- $\sigma = 1.0$

Os parâmetros do algoritmo *Horn & Schunck* são descritos na lista abaixo:

- $\alpha = 15$
- $\varepsilon = 0.5$

O hardware que executa os testes são os mesmos que a obra utiliza, um *Raspberry Pi* com 900MHz de frequência de clock. A memória da *Raspberry Pi* de 1 gigabyte.

Tabela 4.9: Benchmark dataset próprio. Processado Raspberry Modelo 2B Quad-Core ARM cortex-A7 @ 900MHz

Imagem	Expansão Polinomial	Horn & Schunck	Fator Diferença
Sudoeste	182.805	513.705	2,81
Sudeste	183.2466667	515.26	2,81
Noroeste	183.62	515.11	2,81
Nordeste	181.88	510.8	2,81

Na tabela 4.9 estão os tempos dos métodos de expansão polinomial e *Horn & Schunck* em milissegundos. Os únicos valores que não são em milissegundos são os fatores de diferença. A tabela 4.9 mantém-se independente de que conjunto de imagens são executados. Os métodos mantém a diferença de tempo de execução entre os algoritmos expansão polinomial e *Horn & Schunck*. O hardware utilizado para esse benchmark é o mesmo da tabela 4.9, uma *Raspberry Pi* Quad-core ARM cortex-A7 900MHz e 1GB de RAM.

Tabela 4.10: Benchmark dataset próprio, *Horn & Schunck* paralelizado. Processado Raspberry Modelo 2B Quad-Core ARM cortex-A7 @ 900MHz

Imagem	Expansão Polinomial	Horn & Schunck	Fator comparação
Sudoeste	175	96	0,54
Sudeste	174	95	0,54
Noroeste	184	100	0,54
Nordeste	180	95	0,52

A Tabela 4.10 descreve os resultados utilizando o *dataset* proposto para interação da obra. Os tempos na tabela são em milissegundos. O fator de comparação é o quociente entre o tempo da implementação *Horn & Schunck* paralelizado e tempo do método expansão polinomial. Compara-se também em formato de tabela as duas implementações do método *Horn & Schunck*. A Tabela 4.11 descreve os resultados.

Tabela 4.11: Benchmark dataset próprio, *Horn & Schunck* paralelizado e *Horn & Schunck* não paralelizado (serial). Processado Raspberry Modelo 2B Quad-Core ARM cortex-A7 @ 900MHz

Imagem	Horn & Schunck Serial	Horn & Schunck Paralelo	Fator comparação
Sudoeste	513,705	96	0,18
Sudeste	515,26	95	0,18
Noroeste	515,11	100	0,19
Nordeste	510,8	95	0,18

A tabela 4.11 mostra os resultados entre as duas implementações *Horn & Schunck* e as compara. O tempo é novamente em milissegundos. O fator de comparação é quociente entre o a implementação paralela de *Horn & Schunck* e a não paralela (serial). Analisa-se as duas tabelas 4.10, 4.9 e 4.11. Analisando os resultados exibidos pelas tabelas 4.9, 4.10 e 4.11 é observado alguns aspectos. A implementação *Horn & Schunck* não paralela possui um tempo de resposta com um fator de diferença de 2,81 em relação ao tempo da expansão polinomial. A implementação *Horn & Schunck* paralelizada possui um fator de diferença 0,53 em relação ao tempo da expansão polinomial. As implementações *Horn & Schunck* paralelizada e não paralelizadas possuem um fator de diferença de 0,19. Comparando todas as implementações pode se observar que a implementação do *Horn & Schunck* paralelizada é a com melhor resultado. Terminado os resultados de tempo de processamento é mostrado validação dos algoritmos.

4.1.2 Verificação e validação dos algoritmos de fluxo ótico

A forma de analisar validação é o uso de um *dataset* próprio e gerar os próprios resultados visualmente. A Figura 4.8 é gerada utilizando o método *Horn & Schunck*. Os parâmetros utilizados para se gerar o resultados são $\alpha = 15$ e $\epsilon = 0.5$. Portanto utilizando o algoritmo *Horn & Schunck* nos frames da B, o *frame* anterior, e C, o *frame* atual, na Figura 4.8 criamos a imagem A na Figura 4.8. Pintamos onde foi estimado um fluxo ótico maior que um limite estabelecido. O limite considerado para criar a imagem é pelo fato que uma das componente de $\vec{h} = (u, v)$ são maiores em módulo que 0.03 *pixels*.



Figura 4.8: Corretude utilizando *dataset* próprio. A: O fluxo calculado pintado com cores correspondentes as direções na imagem. B: Segundo frame. C: Primeiro frame. As cores indicam a direção de acordo com o círculo na imagem

Analisa-se a coloração da Figura 4.8 C e conclui-se que a implementação do algoritmo *Horn & Schunck* é eficaz. Outra conclusão é que o fluxo estimado se concentra nas bordas dos objetos, porém é suficiente para se utilizar como detecção de movimento. Acredita-se que a implementação poderia ser melhorada nesse quesito, porém o tempo de resposta pode ficar mais lento. Terminada a apresentação de validação da obra descrito o resultado da experiência de interação da obra.

4.1.3 Interatividade da Obra

A interatividade da obra foi limitada por alguns fatores. O ambiente celular era relativamente pequeno, a matriz de 5 por 10 *LEDs* tem o tamanho inteiro da estrutura de madeira. O ambiente celular ter o tamanho apenas da estrutura de madeira faz com que a mudança de resolução tenha mais erro. A resolução de 160 por 120 da câmera para um ambiente de 5 por 10 *LEDs* cria um pequeno erro perceptível, porém não inviabiliza se o movimento for espaçado. O espaçamento citado é no sentido horizontal da imagem ou seja, a interação é mais perceptível se o interator se movimentar um ou dois passos de distância. A câmera capta movimento em uma área significativamente que o ambiente consegue representar. Nesse sentido a obra teve interatividade bem responsiva. O problema maior é o ambiente celular ser pequeno para ser mais responsivo. Explorado os resultados da interação da obra é descrito como foi a apresentação da obra para o público na oportunidade do encontro de arte e tecnologia edição 16.

4.2 Apresentação da Obra

A obra foi participante do encontro de arte e tecnologia *ART16* [33]. O encontro de arte e tecnologia *#ART16* ocorreu entre os dias 1 de setembro e de 29 setembro de 2017 no Museu Nacional da República em Brasília. As fotos da obra sendo expostas estão nas Figuras 4.9 e 4.10.



Figura 4.9: Fazendo a instalação da obra e iluminação.



Figura 4.10: A obra montada de frente.

A obra foi apresentada conjuntamente com outras várias obras do encontro. O encontro obteve uma quantidade estimada de 120 de assinaturas e um número maior de visitantes. A comprovação de participação do evento está no Anexo V. Esses números indicam apenas uma semana de exposição. O Art16 expôs no Museu Nacional da República durante o mês de setembro inteiro. Portanto houve bastante oportunidade de aproximação do público e a ciência da computação. A obra pode ser vista em funcionamento com vídeos através do link <https://www.youtube.com/watch?v=uIUSxkKMSUQ>.

Capítulo 5

Conclusões

Esse capítulo descreve as conclusões e finaliza o texto desse trabalho de graduação. A Seção 5.1 descreve uma análise levando em conta todas as análises feitas no último capítulo. A Seção 5.2 descreve ideias para explorar mais a área e outras propostas para trabalhos.

5.1 Conclusões

O resultado de todo o trabalho foi o proposto, uma obra interativa de arte computacional que utiliza conceitos computacionais. A obra de arte utiliza algoritmos de visão computacional que aproxima o público dos conceitos de computação. Nesse trabalho foi utilizado vários conceitos de visão computacional, placas de circuito impresso, plataformas de desenvolvimento e desempenho de algoritmos. Portanto foram vários conceitos que estavam próximos do público. O projeto *Auravitallis* foi exibido em um encontro de arte e tecnologia no museu com número de visitas significativas, portanto houve visibilidade significativa em um público que não necessariamente está incluído na ciência da computação. As duas versões da obra foram expostas, a primeira versão e a versão que consiste esse trabalho de graduação.

Existiram dificuldades principalmente na produção da placa de circuito impresso. O processo de fabricação da placa foi difícil por falta de experiência com placas de circuito impresso. Outro fator que dificultou foi não ter disponíveis ferramentas melhores para a fabricação. Outra dificuldade que existiu foi encontrar uma interação que seja intuitiva para o interator. Porém a interação de detecção de movimento ficou responsiva na proposta dada sendo uma melhor experiência para o interador se o alcance da câmera seja marcada no ambiente para um outro reforço visual para o interador. A interação ser de detecção de movimento foi melhor alternativa dado os resultados de correte obtidos no instituto *Middlebury*, correte pelo *dataset* próprio e a performance da *Opencv* é a implementação *Horn & Schunck* proposta. A implementação possui menor tempo de resposta para plataformas de desenvolvimento. O custo de uma plataforma de desenvolvimento é relativamente proibitivo. Portanto quanto menor tempo de resposta mais

barato fica para implementação em outros projetos. Porém a implementação vem com um detalhe importante, a plataforma de desenvolvimento necessita possuir processamento paralelo. A implementação proposta poderia ser melhor otimizado, ter fluxo ótico mais correto. Mesmo a implementação não possuir o fluxo ótico mais correto ela tem resultado suficiente para interações da mesma natureza. Outro resultado é a implementação para o algoritmo *Horn & Schunck* para a ferramenta *Processing* na versão *Java*. Essa implementação *Horn & Schunck* pode ser utilizada para outras aplicações porém necessita de ser alterada para programação paralela para melhor tempo de resposta.

5.2 Trabalhos Futuros

Pode ser utilizado o circuito e toda estrutura as classes que calculam o fluxo ótico e retornam um fluxo para outras formas de interação e representações nesse ambiente. Um exemplo de uma interação que pode ser feita futuramente seria uma representação do jogo *snake* onde a detecção de movimento poderia gerar alimentos no ambiente. Outro aspecto a ser explorado pode ser a implementação de redes neurais no comportamento da célula para a tarefa das células se alimentarem. Outro trabalho futuro seria a implementação proposta em formato de biblioteca. Otimização da implementação proposta seria interessante. Outra oportunidade é a continuação de trabalho em parceria com o Medialab/UnB com outras poéticas porém utilizando a mesma implementação do *Horn & Schunck* para a ferramenta *Processing*.

Referências

- [1] DanaH.Ballard Christopher M. Brown. *Computer Vision*. PrenticeHall, Inc., Englewood Cliffs, New Jersey, first edition, 1982. x, 6
- [2] Jorg Rett and Jorge Dias. Autonomus robot navigation - a study using optical flow and log-polar image representation. *Ciência e a Tecnologia Grant 12956/2003*, mar 2003. x, 6, 7
- [3] Daniel Kondermann Enric Meinhardt-Llopis, Javier Sánchez. Horn–schunck optical flow with a multi-scale strategy. *Image Processing On Line*, jun 2012. x, 8, 9, 11
- [4] JP Lewis Stefan Roth Michael Black Richard Szeliski Simon Baker, Daniel Scharstein. Optical flow evaluation, aug 2017. x, 14, 15, 55, 56
- [5] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. xiii, 15, 16, 55, 62
- [6] Simon Monk. *Raspberry Pi Cookbook*. O’Reilly Media, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, second edition, jun 2016. xiii, 17
- [7] T.S. Huang. Computer vision: Evolution and promise. *CERN School of Computing*, 19:46–50, sep 1996. 1, 5
- [8] Bruno Dantas Guedes Elias Melo Filho Filipi Teles da Silva Guilherme Balduino Leandro Ramalho Motta Ferreira Prahada Hargreaves Tainá Luize Ramos Suzete Venturelli, Artur Cabral Reis. *Mutações, Confluências e Experimentações na Arte e Tecnologia*, chapter Auravitalis e Sangeet: na dimensão da desafetação dos objetos. PPG-Arte/UnB, first edition, may 2016. 1, 2, 3, 54
- [9] Oge Marques (auth.) Joel Gibson. *Optical Flow and Trajectory Estimation Methods*. SpringerBriefs in Computer Science. Springer International Publishing, 1 edition, 2016. 6, 8
- [10] Kelson R.T. Aires ; Andre M. Santa; Adelardo A. D. Medeiros. Optical flow using color information: Preliminary results. *SAC’08*, mar 2008. 6
- [11] Song Ho an. 2d convolution separable convolution, aug 2017. 12
- [12] Gunnar Farneback. Two-frame motion estimation based on polynomial expansion. In *SCIA, LNCS 2749*, pages 363–370, Gothenburg, Sweden, June-July 2003. 12, 13, 14

- [13] D. Sun, S. Roth, and M. J. Black. A quantitative analysis of current practices in optical flow estimation and the principles behind them. 106(2):15–137, 2014. 15
- [14] Nikola Davidović Vladimir Milošević Mirjana Maksimović, Vladimir Vujović and Branko Perišić. Raspberry pi as internet of things hardware: Performances and constraints. *International Conference on Electrical, Electronic and Computing Engineering*, 2014. 18
- [15] Processing Foundation. <https://processing.org/>, may 2017. 19
- [16] Open processing. <https://www.openprocessing.org/>, jul 2017. 19
- [17] Processing hardware library. <https://processing.org/reference/libraries/io/index.html>, jul 2017. 19
- [18] Processing Team. Processing gpio digitalwrite. https://processing.org/reference/libraries/io/GPIO_digitalWrite_.html, jul 2017. 20
- [19] OpenCV Community. Opencv about. <http://opencv.org/about.html>, jul 2017. 21
- [20] atduskgreg. Opencv-processing. <https://github.com/atduskgreg/opencv-processing>, jul 2017. 21
- [21] Masashiko Maeda; Kazuya Nagata; Yasutoki Saitou; Taketsugo Ootani; Yuichi Sakon. Printed circuit boards. United States Patent, jun 1988. Patente. 24
- [22] 74hc595 datasheet. <http://pdf1.alldatasheet.com/datasheet-pdf/view/12198/ONSEMI/74HC595.html>, sep 2017. 26
- [23] Clitus Neil D’souza Priyanka Gokarnkar. Comparative study of different moving object detection algorithms and real time implementation using iot based system. *International Journal of Emerging Technology in Computer Science Electronics (IJETCSE)*, 14(2), apr 2015. 26, 27
- [24] Michael Sy Alexander C. Abad Elmer P. Dadios Wilson Feipeng Abaya, Jimmy Basa. Low cost smart security camera with night vision capability using raspberry pi and opencv. In The Institute of Electrical and Electronics Engineers Inc. (IEEE) – Philippine Section, editors, *7th IEEE International Conference Humanoid, Nanotechnology, Information Technology Communication and Control, Environment and Management (HNICEM)*, Hotel Centro, Puerto Princesa, Palawan, Philippines, nov 2014. 26, 27
- [25] Hiroshi Harada. Yukihiro Sugiki, Teruo Yamaguchil. Implementation of optical flow measurement system with an embedded processor. In *15th International Conference on Control, Automation and Systems (ICCAS 2015)*, volume 15, BEXCO, Busan, Korea, 2015. 26, 27
- [26] R.B.Ahmad M.I.Jais D.Shuhaizar I.Iszaidy, R.Ngadiran. Implementation of raspberry pi for vehicle tracking and travel time information system:a survey. *iee*, 2015. 26, 27
- [27] Ming-Hung Lin Yi-You Hou, Sz-Yu Chiou. Real-time detection and tracking for moving objects based on computer vision method. In *2nd International Conference on Control and Robotics Engineering*, International Conference on Control and Robotics Engineering, 2017. 26, 27

- [28] Manesh Murthi Senthil Kumar Thangavel. A semi automated system for smart harvesting of tea leaves. In *International Conference on Advanced Computing and Communication Systems (ICACCS -2017)*, ,Coimbatore, INDIA, jan 2017. Department of Computer Science and Engineering, Department of Mechanical Engineering,. 26, 27
- [29] Grzegorz Granosik Pawel Smyczynski, Lukasz Starzec. Autonomous drone control system for object tracking. *ieee*, 2016. 26
- [30] Raspberry Team. Raspberry 2 b+, description. <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>, apr 2017. 30
- [31] Eagle overview. <https://www.autodesk.com/products/eagle/overview>, sep 2017. 32
- [32] Dan Shiffman Ben Fryes, Casey Reas. Processing libraries gpio. <https://processing.org/reference/libraries/video/Capture.html>, apr 2017. 46
- [33] Cleomar Rocha Suzete Venturelli. art16brasil, sep 2017. 72

Anexo I

Anexo Geral

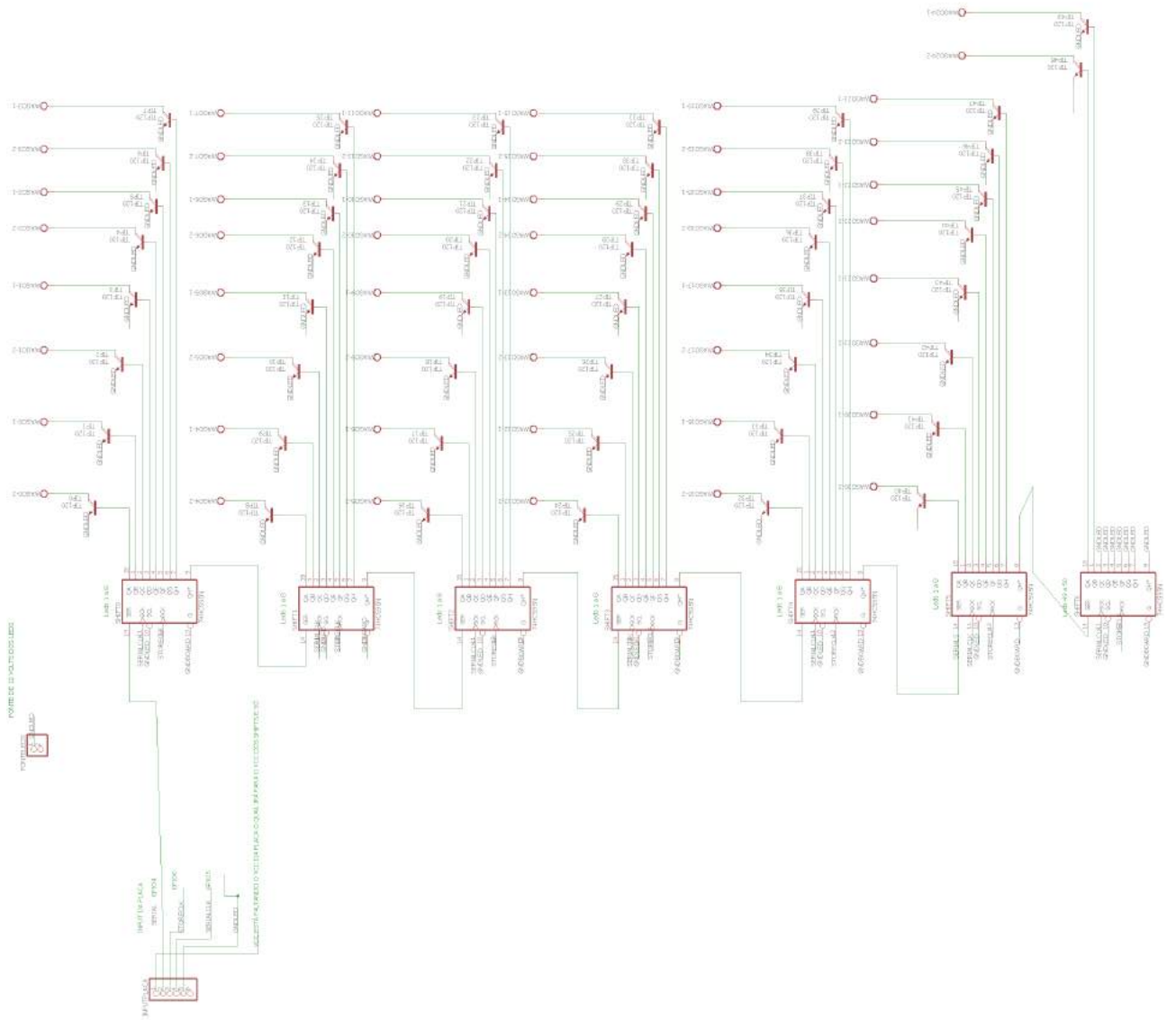


Figura I.1: Esquema lógico da placa

Raspberry Pi 2 Model B (J8 Header)					
GPIO#	NAME			NAME	GPIO#
	3.3 VDC Power	1		2	5.0 VDC Power
8	GPIO 8 SDA1 (I2C)	3		4	5.0 VDC Power
9	GPIO 9 SCL1 (I2C)	5		6	Ground
7	GPIO 7 GPCLK0	7		8	GPIO 15 TxD (UART) 15
	Ground	9		10	GPIO 16 RxD (UART) 16
0	GPIO 0	11		12	GPIO 1 PCM_CLK/PWM0 1
2	GPIO 2	13		14	Ground
3	GPIO 3	15		16	GPIO 4 4
	3.3 VDC Power	17		18	GPIO 5 5
12	GPIO 12 MOSI (SPI)	19		20	Ground
13	GPIO 13 MISO (SPI)	21		22	GPIO 6 6
14	GPIO 14 SCLK (SPI)	23		24	GPIO 10 CE0 (SPI) 10
	Ground	25		26	GPIO 11 CE1 (SPI) 11
30	SDA0 (I2C ID EEPROM)	27		28	SCL0 (I2C ID EEPROM) 31
21	GPIO 21 GPCLK1	29		30	Ground
22	GPIO 22 GPCLK2	31		32	GPIO 26 PWM0 26
23	GPIO 23 PWM1	33		34	Ground
24	GPIO 24 PCM_FS/PWM1	35		36	GPIO 27 27
25	GPIO 25	37		38	GPIO 28 PCM_DIN 28
	Ground	39		40	GPIO 29 PCM_DOUT 29

Attention! The GPIO pin numbering used in this diagram is intended for use with WiringPi / Pi4J. This pin numbering is not the raw Broadcom GPIO pin numbers.

<http://www.pi4j.com>

Figura I.2: Tabela relação pinagem *RaspberryPI 2B*

Anexo II

Anexo Dataset KITTI



Figura II.1: Imagem 0 1242 x 375



Figura II.2: Imagem 1 1242 x 375



Figura II.3: Imagem 2 1242 x 375



Figura II.4: Imagem 3 1242 x 375



Figura II.5: Imagem 4 1242 x 375



Figura II.6: Imagem 5 1242 x 375



Figura II.7: Imagem 6 1242 x 375



Figura II.8: Imagem 7 1242 x 375



Figura II.9: Imagem 8 1242 x 375



Figura II.10: Imagem 9 1242 x 375



Figura II.11: Imagem 10 1242 x 375



Figura II.12: Imagem 11 1242 x 375



Figura II.13: Imagem 12 1242 x 375



Figura II.14: Imagem 13 1242 x 375



Figura II.15: Imagem 14 1242 x 375



Figura II.16: Imagem 15 1242 x 375



Figura II.17: Imagem 16 1242 x 375



Figura II.18: Imagem 17 1242 x 375



Figura II.19: Imagem 18 1242 x 375



Figura II.20: Imagem 19 1242 x 375

Anexo III

Anexo Dataset Middlebury



Figura III.1: Army 584 x 388



Figura III.2: Bakyard 584x388



Figura III.3: Basketball 640x480



Figura III.4: Dumptruck 640x480



Figura III.5: Evergreen 584x388



Figura III.6: Grove 640x480



Figura III.7: Mequon 640x480



Figura III.8: Schefflera 640x480



Figura III.9: Teddy 420x380



Figura III.10: Urban 640x480



Figura III.11: Wooden 640x480

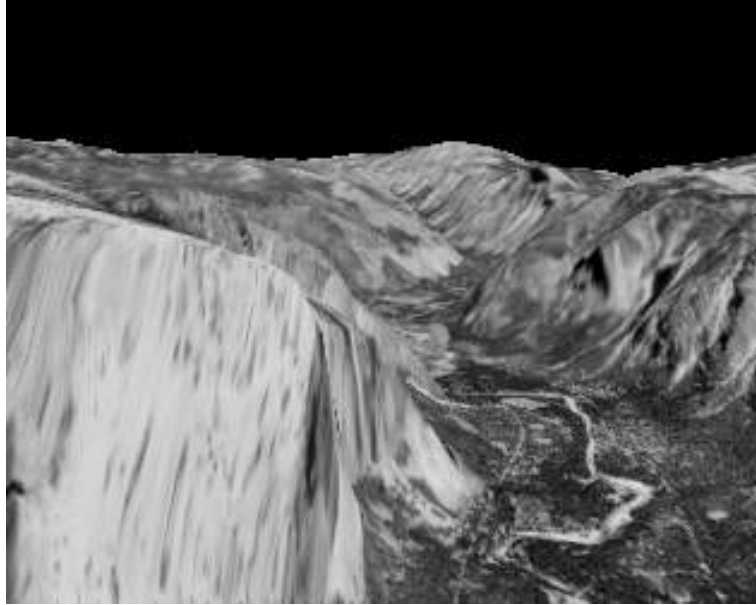


Figura III.12: Yosemite 640x480

Anexo IV

Anexo Interação



Figura IV.1: Imagem Interação Nordeste 160 x 120



Figura IV.2: Imagem Interação Noroeste 160 x 120



Figura IV.3: Imagem Interação Sudeste 160 x 120



Figura IV.4: Imagem Interação Sudoeste 160 x 120

Anexo V

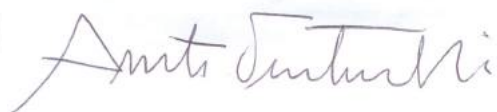
Anexo Apresentacao Museu

16° Encontro Internacional de Arte e Tecnologia (#16.ART): artis intelligentia-imaginar o real e 2° Colóquio Retina. Internacional: film-still-life

atestado

Vimos por meio deste atestar que **Leandro Ramalho Motta Ferreira** participou da exposição **#2Retina**, com a obra **Auravitallis**, assim do **16° Encontro Internacional de Arte e Tecnologia (#16.ART): artis intelligentia-imaginar o real e 2 Colóquio Retina.Internacional: film-still-life**, em setembro de 2017, no Museu Nacional da República de Brasília.

Atenciosamente,



Suzete Venturelli
Comissão Organizadora



Cleomar Souza Rocha
Comissão Organizadora

MEDIA
LAB/BR

MEDIA
LAB/UNB



MEDIA
LAB/UFG

UFG



Secretaria de
Cultura



GOVERNO DE
BRASÍLIA



