UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

BIBLIOTECA PARA RECONSTRUÇÃO DE RELEVOS

KEVIN STORTZ

BLUMENAU
2017

KEVIN STORTZ

BIBLIOTECA PARA RECONSTRUÇÃO DE RELEVOS

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

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

BLUMENAU
2017

BIBLIOTECA PARA RECONSTRUÇÃO DE RELEVOS

Por

KEVIN STORTZ

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

Presidente:	Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB
Membro:	Prof. Leandro Augusto Frata Fernandes, Dr. – UFF
Membro:	Prof. Aurélio Faustino Hoppe, M.Sc. - FURB

Blumenau, 12 de dezembro de 2017

Dedico este trabalho a comunidade acadêmica com o intuito de que ele sirva como base para futuros estudos correlatos.

AGRADECIMENTOS

Agradeço a minha mãe, Mariana por todos os incentivos e cobranças para que eu alcance meus objetivos e ao meu pai, Frank por me incentivar a sempre buscar o que me faz feliz.

A minha namorada, Danieli pelos incentivos, cobranças e pelo auxílio na correção ortográfica.

Ao meu orientador, Dalton pelo apoio e confiança no desenvolvimento do projeto.

Ao Dr. Leandro Augusto Frata Fernandes, especialista da área, que forneceu o conteúdo necessário para o desenvolvimento desse projeto.

“The Man who says he can, and the man who says he cannot. Are both correct”

Confucius

RESUMO

O reconhecimento de relevos e a reconstrução 3D apresentam processamentos com alto grau de complexidade computacional, por terem que realizar diversos algoritmos de correlação sobre duas imagens distintas. Conceitos como a Geometria Epipolar visam diminuir a complexidade de utilização e aprimorar a definição para a correlação de duas visualizações distintas do mesmo cenário. É utilizado um modelo de câmera *pinhole,* sendo feito o processo de calibração da câmera, a fim de definir a matriz de parâmetros intrínsecos, o coeficiente de distorção radial e a matriz de parâmetros extrínsecos, e realizar a calibração do projetor, sendo assim possível utilizá-lo como uma câmera inversa. É definido também a matriz fundamental que encapsula a Geometria Epipolar, obtida através da etapa de calibração estéreo. Tendo com a Geometria Epipolar o mapeamento dos pixels de duas visualizações da mesma cena. Sendo possível assim emitir um padrão de luz estruturado sobre o ambiente, percebendo a distorção deste padrão e reconhecendo a profundidade para a reconstrução 3D. Por fim, tem-se as etapas para realizar a correlação entre pixels baseando-se na distância euclidiana das cores dos pixels e sendo apresentado um mapa de disparidade para analisar os resultados das correlações definidas. Como resultados tem-se os mesmos resultados obtidos no processo de calibração do software utilizado como referência e é possível identificar um erro ao percorrer os pixels da linha epipolar a partir da imagem gerada do mapa de disparidade.

Palavras-chave: Reconstrução 3D. Geometria epipolar. Calibração câmera-projetor. Calibração estéreo. Correlação de pixels.

ABSTRACT

Terrain recognition and 3D reconstruction are computer processes that have a high degree of computational complexity. Because they need to perform many correlation algorithms between two distinct images. Concepts as Epipolar Geometry aim to decrease the complexity to perform it and improve the definition of correlation from two distinct visualizations of the same scene. It will be used a camera pinhole model, that will be needed the calibration process, to define the intrinsic parameters matrix, the radial distortion coefficient and the extrinsic parameters matrix. Also need to be made the projector calibration, so it can be used as an inverse camera. It will also defined by the stereo calibration process the fundamental matrix that encapsulate the Epipolar Geometry. With the Epipolar Geometry it is possible to map the pixels two distinct views from the same scene. So, it is possible to emit a structured light pattern over the scene and with the camera perceiving the distortion of this pattern over the 3D model, so it will be able to reconstruct it. The last steps consist to correlate the pixels from the two views based in the Euclidian distance from the colors of the two pixels and generating a disparity image to analyses the correlation process. Some results can be observed in the retroprojection errors in calibration process that are the same from library as so from the software used as reference, and in the disparity image generated was observed that can be an error in the epipolar line definition process or the correlation step, because the scene terrain cannot be observed in disparity image.

Key-words: 3D reconstruction. Epipolar geometry. Camera-projector calibration. Stereo calibration. Pixel correlation.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API – Application Programming Interface

CPU - Central Process Unit

FPS – Frames Por Segundo

GPGPU – General-Purpose Graphics Processing Unit

GPU - Graphics Process Unit

SSD - Sum of Squared Diferences

SUMÁRIO

1 INTRODUÇÃO

Segundo PENA (2017, p. 1, grifo do autor) “O **relevo** é a parte superficial da litosfera (camada sólida da Terra). É onde as transformações geológicas se expressam mais nitidamente, sendo também o local de habitação do ser humano e da maior parte dos animais terrestres.” Tem-se então que os relevos são as formas físicas da superfície terrestre e podem ser divididas em: montanhas, planaltos, planícies e depressões (PENA, 2017). Existem dois métodos que podem ser utilizados para representar a altura de relevos em mapas: Carta Hipsométrica e Curvas de Nível (MENDONÇA, 2007).

Cartas Hipsométricas são utilizadas para representação de relevos em mapas de pequena escala e grandes áreas, utilizando cores para representação de diferentes altitudes. Enquanto as Curvas de Nível são utilizadas em mapas de grande escala, separando as alturas dos relevos por linhas entre intervalos fixos (MENDONÇA, 2007).

A Universidade Regional de Blumenau, desenvolve um projeto que consiste em exibir a Carta Hipsométrica e as Curvas de Nível sobre um relevo moldado com areia. O projeto faz utilização de uma aplicação desenvolvida pela universidade da Califórnia em Davis que através do Kinect consegue realizar o processo de detecção e reconstrução do relevo em tempo real. Possibilitando assim a visualização das cartas sobre a areia conforme o cenário é alterado (BOHN, 2017). Tendo então a dependência da biblioteca desenvolvida pela universidade da Califórnia e a utilização do Kinect para reconhecimento da profundidade. Sendo assim, poderia ser utilizada duas câmeras para realizar o processo de reconstrução 3D e visualizar as cartas da mesma forma como é feito atualmente, removendo ainda a dependência da biblioteca externa e o equipamento Kinect.

Tendo a necessidade de ter uma aplicação em tempo real para mapear as alterações no relevo nas imagens da Carta Hipsométrica e Curvas de Nível na disponibilização de um dataset para plataformas como o Unity. Segundo Zaaijer (2013, p. 32), a aplicação deve manter uma taxa de 30 a 60 quadros por segundo (Frames Per Second - FPS) para ser considerada em tempo real. Sendo assim, a biblioteca deve utilizar processamento paralelo para realizar os cálculos necessários na reconstrução 3D. Sobre o processamento paralelo, Fernandes (2017a) diz que se tem a necessidade de utilizar o processamento disponível pela unidade de processamento gráfico (Graphics Programming Unit - GPU) no processo de triangulação da imagem, que tem seu desenvolvimento facilitado através da Application Programming Interface (API) CUDA, para comunicação com a GPU.

Para a reconstrução do ambiente em um modelo 3D. Segundo Fernandes (2005, p. 6), a extração de informações tridimensionais, a partir de imagens, tem sido um assunto amplamente estudado no ramo de Visão Computacional (VC). Geralmente essa extração utiliza o projetor para apresentar imagens, e uma câmera para captura do estado atual do ambiente. Dentre alguns conceitos da VC que podem ser aplicados, conforme estudo apresentado por Fernandes (2005, p. 10), a técnica de reconstrução por triangulação ativa provavelmente é o método mais antigo e amplamente estudado para detecção de profundidade, utilizando a Geometria Epipolar para a reconstrução 3D (HARTLEY; ZISSERMAN, 2004, p. 239).

Para que seja possível realizar a reconstrução 3D do ambiente é necessário realizar a calibração e configuração das câmeras e do ambiente estéreo (ZHANG, 2000, p. 2). O ambiente estéreo consiste na utilização de duas câmeras observando o mesmo ambiente, a fim de realizar a triangulação entre os objetos observados (THE MATHWORKS, 2017). Segundo Moreno e Taubin (2012, p. 1, tradução nossa) “Um par projetor e câmera podem trabalhar como se fosse um ambiente estéreo”, sendo que, ainda segundo Moreno e Taubin (2012, p. 1, tradução nossa) “Neste sistema o projetor é identificado como uma câmera inversa”.

Segundo Hartley e Zisserman (2004, p. 153, tradução nossa) “Uma câmera é um mapeamento entre o mundo 3D (objeto espacial) e uma image 2D”. Tendo a necessidade realizar a etapa de calibração para a definição da matriz das câmeras, que definem a rotação e translação necessária para correlacionar um ponto 3D na imagem 2D (HARTLEY; ZISSERMAN, 2004, p. 152). Tendo então a Geometria Epipolar como forma de representação da geometria de mapeamento entre duas câmeras, sendo que a matriz fundamental encapsula a geometria intrínseca, possibilitando realizar a correlação entre dado pixel da primeira câmera com o correlato na linha epipolar da segunda câmera (HARTLEY; ZISSERMAN, 2004, p. 239). Um mapa de disparidade pode ser gerado para validar a correlação dos pixels. O mapa que consiste em verificar a distância euclidiana entre os pixels correlacionados. Normalizando a distância entre 0 e 255 a fim de gerar uma imagem em escala de cinza, que apresenta a validação visual da profundidade (FERNANDES, 2017d).

Segundo Hartley e Zisserman (2004, p. 310, tradução nossa) “É assumido que existem erros na medição das coordenadas dos pontos”, ainda segundo Hartley e Zisserman (2004, p. 310, tradução nossa) “Com base nessas circunstâncias a triangulação gulosa por projeção inversa de raios dos pontos medidos falhará, pois, os raios geralmente não terão intersecção. Sendo necessário estimar uma melhor solução para o ponto no espaço 3D.” Sendo assim,

antes de realizar a triangulação, torna-se necessário realizar a correção da geometria para ter os pontos que se interseccionarão tornando possível gerar um modelo 3D.

Conforme apresentado, é desenvolvida uma biblioteca que fará a detecção e reconstrução de relevos por meio da Geometria Epipolar e possibilita a utilização de programação paralela na GPU para otimização de cálculos em imagens. Como resultado será visualizada a calibração do ambiente estéreo, bem como a correlação dos pixels com as linhas epipolares e a exibição do mapa de disparidade da correlação dos pixels.

1.1 OBJETIVOS

O objetivo deste trabalho é criar uma biblioteca para detecção e reconstrução de relevos utilizando os conceitos de Geometria Epipolar.

Os objetivos específicos são:

a) criar um padrão para ser projetado na superfície a ser utilizada na reconstrução;
b) realizar a calibração do ambiente estéreo, gerando os dados necessário para a Geometria Epipolar;
c) realizar a correlação dos pixels entre as duas visualizações do ambiente;
d) gerar um mapa de disparidade para validar a correlação dos pixels.

1.2 ESTRUTURA

Na fundamentação teórica serão analisados três trabalhos com funcionalidades similares às que foram apresentadas acima. Também serão apresentados conhecimentos básicos da área topológica, Geometria Epipolar e computação que serão utilizados para guiar o desenvolvimento da aplicação.

No desenvolvimento serão apresentados os detalhes técnicos da aplicação, incluindo especificações, técnicas, ferramentas e os diversos algoritmos utilizados durante o processo. Por último, serão apresentados os resultados dos testes e as conclusões que podem ser tiradas, assim como possíveis extensões do projeto.

2 FUNDAMENTAÇÃO TEÓRICA

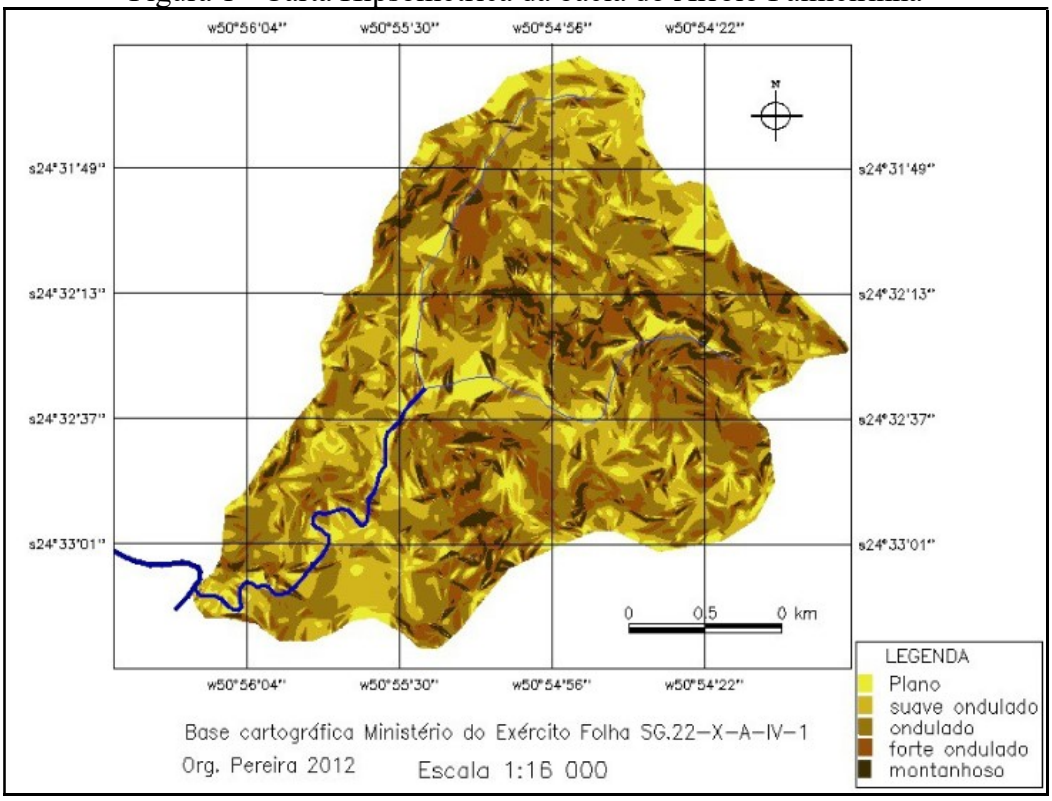
Este capítulo está organizado em quatro seções. A seção 2.1 aborda as cartas topográficas e suas representações em imagens planas. A seção 2.2 descreve o processo de reconstrução 3D. Por fim, a seção 2.4 apresenta os trabalhos correlatos.

2.1 CARTAS TOPOGRÁFICAS

Para representar as diferentes altitudes da superfície terrestre em mapas são utilizados dois métodos: Carta Hipsométrica e Curvas de Nível. Para mapas de menor escala e mapeamento de grandes áreas é comumente utilizado a representação de Carta Hipsométrica. Já em mapas de maior escala são utilizadas as Curvas de Nível, entretanto tanto a hipsométrica como Curvas de Nível poderiam ser empregados em um mesmo mapa (MENDONÇA, 2007).

A Carta Hipsométrica é utilizada para representar as diferenças de altitude em mapas, pois facilita a compreensão dos dados utilizando um padrão de cor. Um exemplo de uso da Carta Hipsométrica pode ser observado no trabalho de Pereira e Thomaz (2013, p. 3494), onde tem-se o estudo do impacto da declividade das encostas. Pode ser visto na Figura 1, segundo Pereira e Thomaz (2013, p. 3498), “que as maiores declividades concentram-se nas áreas com altitudes variando entre 780 e 900 metros, esta faixa de altitude seria uma área de ‘quebra’ de declive da bacia, e divisor das formações geomorfológicas”.

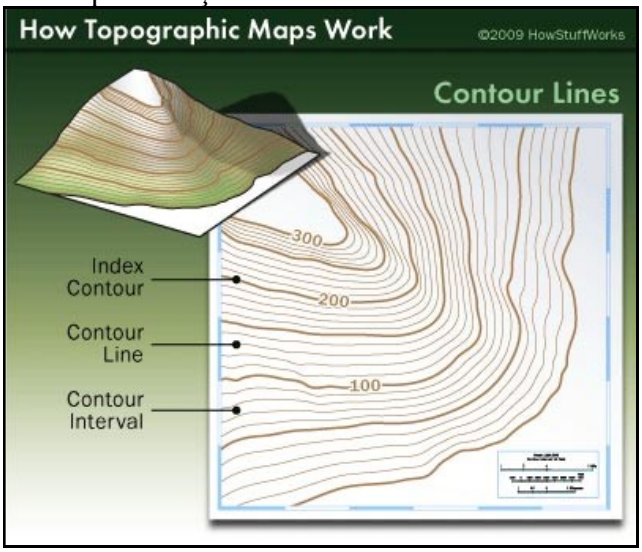
Figura 1 - Carta Hipsométrica da bacia do Arroio Palmeirinha



Fonte: Pereira e Thomaz (2013, p. 3499).

As Curvas de Nível possibilitam a visualização plana da inclinação dos relevos. Conforme Figura 2, o relevo é separado entre linhas de igual altitude, sendo que as linhas principais (índices) possuem igual distância entre si. Esta forma de apresentação possibilita uma compreensão tridimensional a partir de uma imagem plana (MENDONÇA, 2007). Para aprimorar a leitura do mapa podem ser definidas linhas de indexação, exemplo: a cada 100 metros a linha recebe um destaque perante as outras (RONCA, 2009).

Figura 2 - Representação de relevo utilizando Curvas de Nível



Fonte: Ronca (2009).

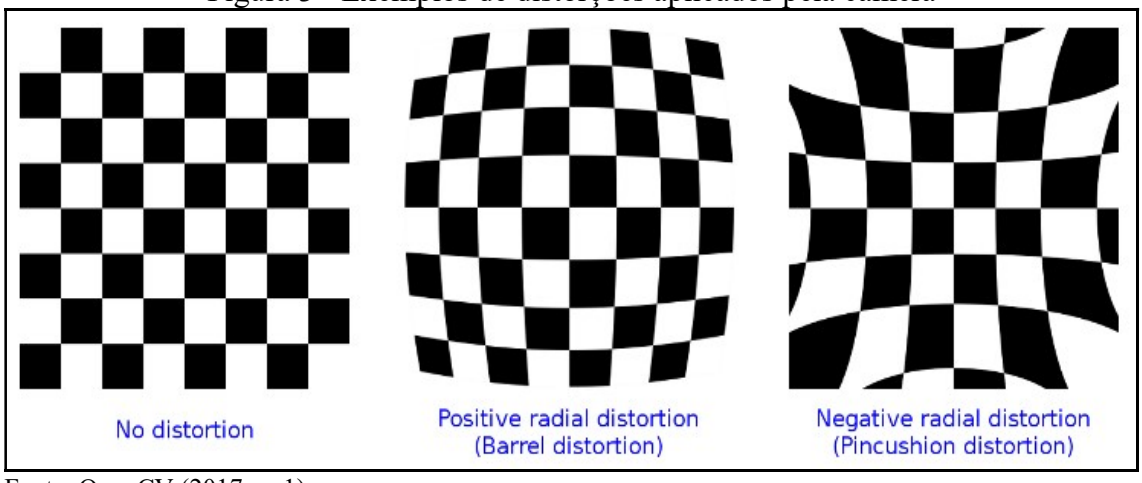
2.2 RECONSTRUÇÃO 3D

Nesta seção são descritas as etapas utilizadas neste projeto para realizar a reconstrução 3D de uma cena. Na seção 2.2.1 é apresentado o processo de calibração da câmera e do projetor a partir da imagem vista pela câmera. A seção 2.2.2 é descrita a Geometria Epipolar apresentando seus conceitos e fórmulas que possibilitam a reconstrução. Na seção 2.2.3 é contextualizado o processo de triangulação ativa que permite a percepção da profundidade das imagens 2D e geração da nuvem de pontos.

2.2.1 CALIBRAÇÃO DA CÂMERA

Segundo Zhang (2000, p.2, tradução nossa), "A calibração da câmera é uma etapa necessária na visão computacional 3D para extração de informações métricas de imagens 2D." A calibração visa definir a matriz de parâmetros intrínsecos que contém os dados do modelo da câmera, como: o centro da câmera e a distorção focal da câmera. Além de definir a matriz de parâmetros intrínsecos da câmera, a calibração pode definir do efeito apresentado na Figura 3, observa-se que as câmeras possuem uma distorção enquanto capturam as imagens. Na reconstrução 3D para manter a mesma proporção entre os elementos observados tem-se a necessidade de observar a imagem como um plano sendo assim, necessária a correção da distorção da câmera (OPENCV, 2017, p. 1).

Figura 3 - Exemplos de distorções aplicados pela câmera



Fonte: OpenCV (2017, p. 1).

As técnicas de calibração de câmera podem ser divididas em duas categorias: calibração fotogramétrica e auto calibração. A calibração fotogramétrica consiste em observar com a câmera um objeto que possui sua geometria 3D conhecida, tendo assim como verificar a distorção causada pela câmera. A auto calibração não utiliza nenhum objeto para a calibração, ela consiste apenas em observar a cena estática tirando fotos e caso sejam

definidos parâmetros internos fixos, com 3 imagens já é possível recuperar os parâmetros internos e externos (ZHANG, 2000, p. 2).

O método de calibração proposto por Zhang (2000) consiste em observar uma imagem contendo um padrão planar em no mínimo duas orientações distintas, sendo que, tanto a câmera quanto o padrão podem ser rotacionados para a obtenção das novas imagens (ZHANG, 2000, p.3). Segundo Zhang (2000, p.3, tradução nossa), "o método está entre o método de calibração fotogramétrica e do auto calibração, pois são utilizadas métricas 2D ao invés de 3D ou puramente implícitas." Através da visualização do plano é possível determinar os parâmetros intrínsecos e os parâmetros extrínsecos para a câmera (ZHANG, 2000, p. 3).

A partir da definição dos parâmetros intrínsecos e extrínsecos torna-se capaz a definição da fórmula do modelo de câmera *pinhole*. Conforme visto na Figura 4, a segunda matriz representa os parâmetros intrínsecos sendo uma matriz 3 X 3, representa os valores internos de cada câmera, com `fx` e `fy` representando a distorção focal de cada câmera, e `cx` e `cy` representando o ponto principal da câmera, ou ponto central da câmera. A terceira matriz é a dos parâmetros extrínsecos sendo uma matriz 3 X 4, onde nela estão contidos a rotação e a translação necessária para mapear a transformação de um ponto 2D da imagem da câmera para o seu respectivo ponto 3D na cena (ZHANG, 2000, p. 3-4).

Figura 4 - Matrizes do modelo de câmera *pinhole*

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Fonte: OpenCV (2017, p.1).

Na Figura 5 é apresentada a fórmula do modelo de câmera *pinhole* das matrizes da Figura 4, que faz o mapeamento de um ponto na imagem da câmera com o mesmo ponto 3D na cena (HARTLEY; ZISSERMAN, 2004, p.153). Tendo `m` como um ponto na imagem tal que `m = [u, v, 1]`, onde `u` e `v` representam um ponto `x` e `y` da imagem da câmera e o ponto 3D respectivo `M = [X, Y, Z, 1]`. É identificado `A` como sendo a matriz dos parâmetros intrínsecos e `R` e `t` sendo a matriz dos parâmetros extrínsecos (ZHANG, 200, p. 3).

Figura 5 - Fórmula do modelo de câmera *pinhole*

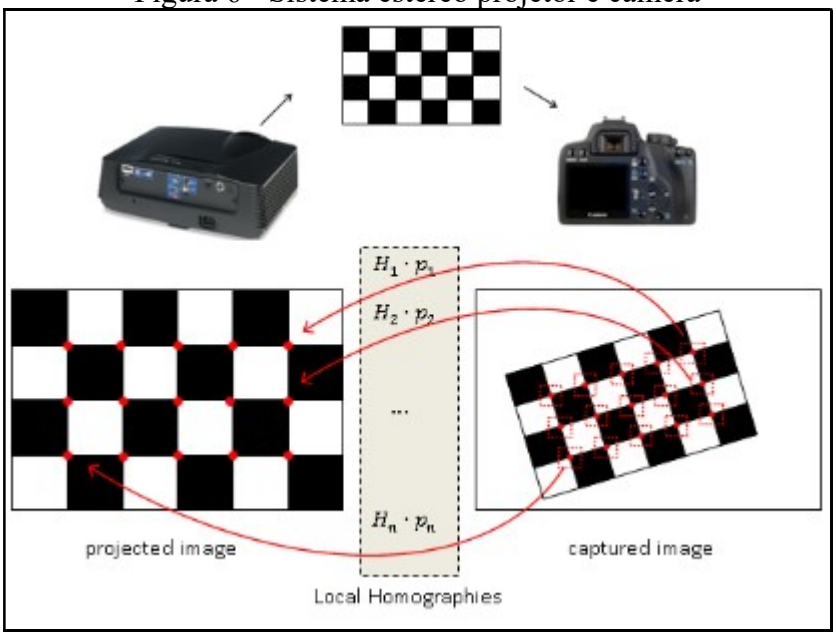
$$s\tilde{\mathbf{m}} = \mathbf{A}\begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}\tilde{\mathbf{M}},$$

Fonte: Zhang (2000, p.3).

A etapa de calibração da câmera apresentada será utilizada em conjunto com um projetor formando assim um sistema de reconstrução com múltiplas visões do ambiente, que

segundo Hartley e Zisserman (2004, p. 239, tradução nossa) “A Geometria epipolar é uma geometria projetiva intrínseca entre duas visualizações.” Ainda segundo Moreno e Taubin (2012, p. 1, tradução nossa) “o conjunto entre uma câmera e projetor podem ser usados como um sistema estéreo”. O sistema estéreo na Figura 6 representa a etapa de calibração entre as duas visualizações (MORENO; TAUBIN, 2012, p. 1).

Figura 6 - Sistema estéreo projetor e câmera



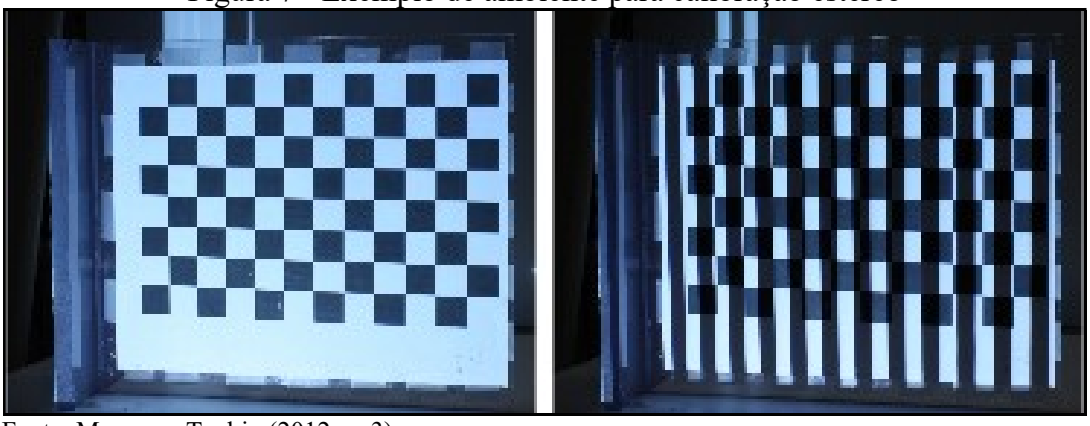
Fonte Moreno e Taubin (2012, p. 1).

O processo de calibração deste sistema estéreo consiste no algoritmo apresentado por Moreno e Taubin (2012). O trabalho proposto realiza o processo de calibração levando em consideração a distorção radial tanto da câmera quanto do projetor. Segundo Moreno e Taubin (2012, p. 1, tradução nossa) “o procedimento de calibração deve ser adaptado levando em consideração que o projetor não pode identificar as coordenadas dos pixels 3D na imagem projetado assim como a câmera faz.” Sendo assim, o processo de calibração teve que ser adaptado para que fosse possível estimar as coordenadas dos pontos calibrados pela câmera na imagem projetada (MOREN; TAUBIN, 2012, p. 1).

Para realizar a calibração do projetor o algoritmo utiliza uma estrutura de luz conhecida que é projetada sobre a imagem do padrão. Na Figura 7 é apresentado um exemplo de configuração de ambiente para calibração estéreo. Primeiramente, é realizada a calibração da câmera tratando a quantidade de luz recebida pela imagem, identificando os quadrados do tabuleiro de xadrez. Tendo identificado a posição dos quadrados, pode ser feita a classificação das linhas e colunas onde o projetor está aplicando a luz estruturada. A próxima etapa consiste em tirar fotos alterando o padrão da luz estruturada, utilizando o tamanho previamente encontrado do tabuleiro de xadrez podendo assim ser definida uma homografia que realiza a

conversão da coordenada da câmera para o projetor. Tendo definido essa homografia é realizada a etapa de conversão dos quadrados observados pela câmera para coordenadas do projetor. Por último são corrigidas as coordenadas dos pontos para o ambiente do mundo, afim de aplicar o processo de calibração de Zhang (2000), tanto para câmera quanto para o projetor. Sendo assim, também é possível realizar a calibração estéreo do ambiente gerando a matriz de R e T dos pontos da câmera para o projetor, e a matriz fundamental da Geometria Epipolar (MORENO; TAUBIN, 2012, p. 5–6).

Figura 7 - Exemplo de ambiente para calibração estéreo



Fonte: Moreno e Taubin (2012, p. 3).

2.2.2 GEOMETRIA EPIPOLAR

Segundo Hartley e Zisserman (2004, p. 239), o objetivo da Geometria Epipolar é permitir a intersecção entre os pontos dos planos das imagens de duas visualizações. Conforme visto na seção 2.2.1 o ambiente é calibrado para a utilização de uma câmera e um projetor como visualizações estéreo, tendo assim a necessidade de realizar a correspondência dos pixels entre as imagens. Portanto, para cada x na imagem da câmera é necessário encontrar o x′ correspondente na projeção, possibilitando assim definir a profundidade que o X correspondente encontra-se na cena 3D (HARTLEY; ZISSERMAN, 2004, p. 240).

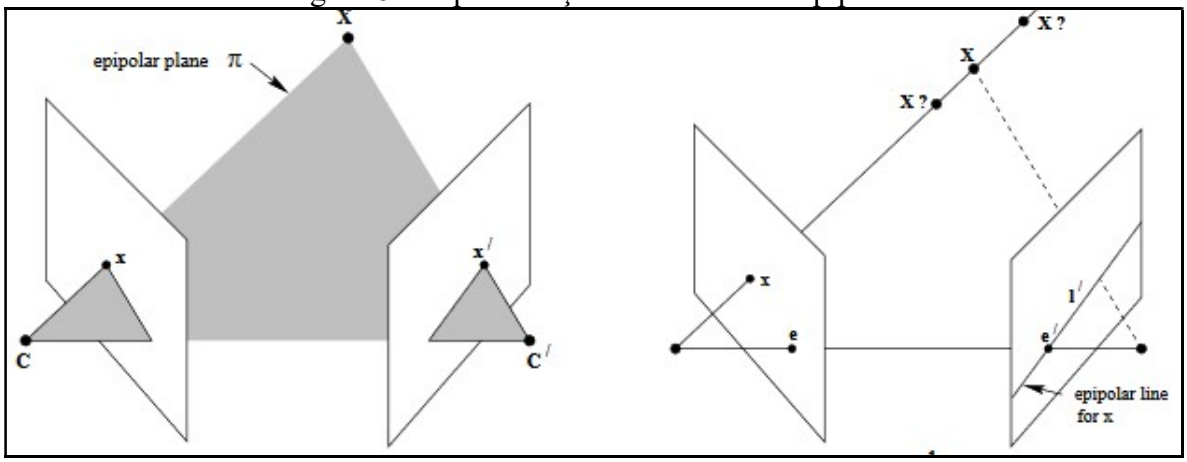
Na Figura 8 é representado como a Geometria Epipolar faz o mapeamento entre os pixels das duas visões. Para o entendimento da geometria Hartley e Zisserman (2004, p. 240-241) apresentam os seguintes conceitos:

a) epípolo: representados por e e e′, são os pontos que geram a interseção dos centros das duas câmeras, a linha que liga os dois epípolos e os dois centros também é conhecida como linha base;
b) plano epipolar: é o plano que contém a linha base, nele estão contidos os pontos entre x, x′ e X;
c) linha epipolar: é a intersecção entre o plano epipolar e o plano da imagem, todas as

linhas epipolares interseccionam com o epípolo;

d) matriz fundamental: é a representação algébrica da geometria epipolar, é uma matriz 3 X 3 com rank 2 e 7 graus de liberdade (HARTLEY; ZISSERMAN, 2004, p. 246).

Figura 8 - Representação da Geometria Epipolar



Fonte: Hartley e Zisserman (2004, p. 240).

O processo de mapeamento dos pixels entre as duas imagens consiste em: dado um ponto `x` na primeira imagem é aplicada a fórmula da Figura 9, resultando na definição de uma linha epipolar denominada `l′` na segunda imagem. Sendo assim é necessário apenas percorrer esta linha para encontrar o ponto `x′` correspondente, tendo `F` como sendo a matriz fundamental calculada na seção 2.2.1 e `x` cada ponto da imagem da câmera (HARTLEY; ZISSERMAN, 2004, p. 240-243).

Figura 9 - Fórmula resultante na linha epipolar

$$\mathbf{l}' = [\mathbf{e}']_{\times}\mathbf{H}_{\pi}\mathbf{x} = \mathbf{F}\mathbf{x}$$

Fonte: Hartley e Zisseman (2004, p.243).

2.2.3 TRIANGULAÇÃO

Para ser realizada a etapa de triangulação entre as duas visualizações da cena identificando assim a profundidade dos elementos, é necessário realizar o processo de correlação entre os pixels. Tendo como resultado a correlação `x` e `x′` apresentados na seção anterior (HARTLEY; ZISSERMAN, 2004, p. 243). Segundo Fernandes (2017b) o processo inteiro consiste em três etapas:

a) identificar as intersecções das linhas epipolares com as bordas da imagem do projetor;

b) percorrer todos os pixels da linha epipolar para identificar a posição do pixel correspondente;

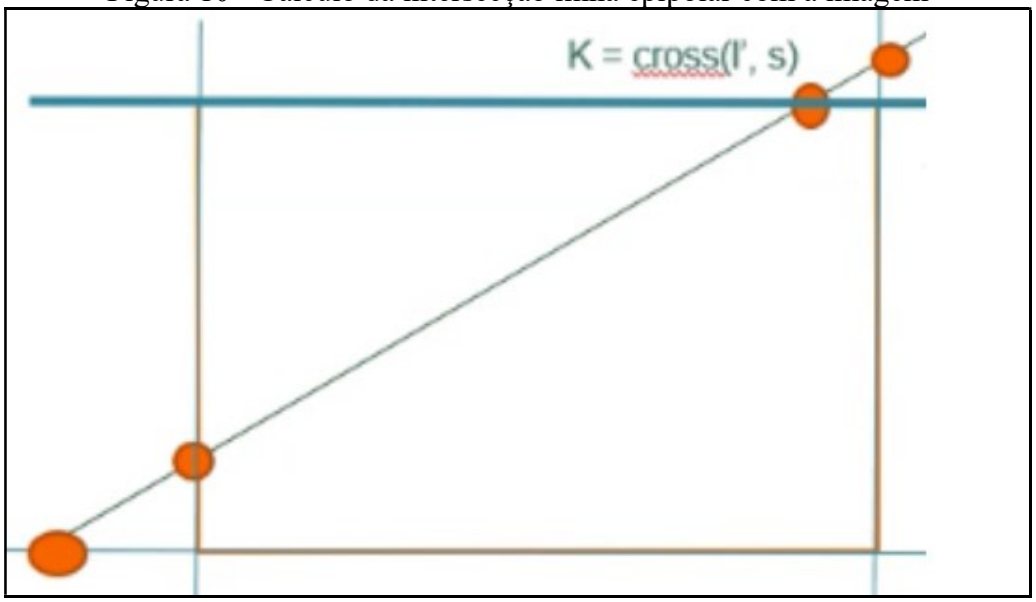
c) corrigir o erro da geometria, possibilitando a aplicação do raio nos dois pixels para identificar a intersecção e consequentemente a posição 3D.

Na primeira etapa é necessário realizar a multiplicação da matriz fundamental com cada pixel da imagem capturada pela câmera para ser gerado a linha epipolar na imagem do projetor. A linha epipolar define uma equação de reta no espaço 2D, tendo ainda impossibilitada a definição dos pontos de intersecção da linha epipolar com as bordas do padrão estruturado projetado (FERNANDES, 2017b).

Para que seja possível identificar explicitamente quais são os pixels da linha epipolar, é possível utilizar o cálculo do produto vetorial entre os 4 pixels dos cantos da imagem e a equação da reta da linha epipolar. Quando aplicado o produto vetorial entre dois pixels tem-se como resultado a equação da reta que atravessa por estes dois pontos. Necessitando assim a aplicação do produto vetorial entre os quatros cantos da imagem do projetor, para que sejam definidas as equações de reta das bordas da imagem (HARTLEY; ZISSERMAN, 2004, p. 23).

Para finalizar o processo da primeira etapa, tendo as equações da reta das bordas e a equação da reta da linha epipolar, deve-se aplicar o produto vetorial entre a linha epipolar com cada equação da borda. O resultado desta equação gerará um ponto de intersecção entre as duas retas. Tendo então as intersecções da linha epipolar com os quatro cantos da imagem. Para transformar a equação da reta em um ponto 2D é preciso normalizar os valores (FERNANDES, 2017b). O processo pode ser visualizado na Figura 10. Para realizar os cálculos do produto vetorial é utilizada a biblioteca Mathfu de matemática desenvolvida pelo Google, que consiste em uma biblioteca *header only* e pode ser utilizada em projetos desenvolvidos na linguagem C++.

Figura 10 - Cálculo da intersecção linha epipolar com a imagem



Fonte: Fernandes (2017b) adaptado pelo autor.

Por fim verifica-se quais pontos que estão mais próximos da borda da imagem, tendo então o ponto inicial e o ponto final da intersecção. Tendo estes dois pontos é possível usar o algoritmo de Bresenham, que define um método para percorrer todos os pontos em uma matriz de duas dimensões dado um ponto inicial e um final (FERNANDES, 2017b).

A segunda etapa define a correlação entre um pixel x com x' percorrendo todos os pixels na linha definida na etapa anterior. O algoritmo utilizado para realizar esta correlação consiste de um algoritmo denominado Sum of Squared Diferences (SSD), adaptado para somar a distância euclidiana das cores, no sistema de cores L*a*b*, comparando o pixel da imagem de referência com todos os pixels da linha epipolar (FERNANDES, 2017c).

Conforme é apresentado na Figura 11 a fórmula para calcular o SSD consiste em realizar o somatório entre a diferença de duas regiões de pixels, neste caso, a diferença entre o *kernel* da imagem do projetor e da câmera. O padrão que é usado no projetor trata-se de uma imagem colorida, sendo assim possível levar em consideração a diferença de crominância entre as cores entre os pares da imagem no sistema de cores L*a*b* (FERNANDES, 2017c).

Figura 11 - Fórmula para realizar a correlação entre pixels em escala de cinza.

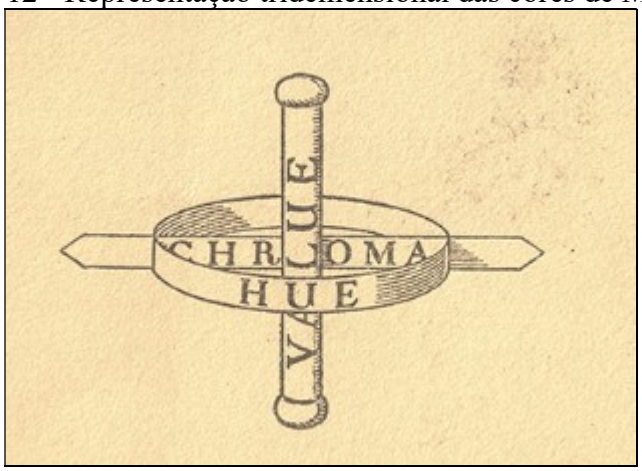
$$\sum_{[i,j]\in R} (f(i,j) - g(i,j))^2$$

Fonte: Collins (2012, p. 26).

Segundo Munsell (1929, p. 9, tradução nossa) “a cor possuí três dimensões, matiz, valor e croma, que descrevem precisamente e totalmente qualquer cor tão facilmente quanto as três dimensões de uma caixa descrevem seu comprimento, largura e espessura.” Na Figura 12 é apresentada a representação tridimensional das cores seguindo a teoria de Munsell.

A Matiz representa a identificação inicial de uma cor para o olho do ser humano, é como identificamos que uma cor verde é verde e não vermelha. O Valor refere-se à quantidade de luz aplicada na cor, pois quanto mais próximo do branco mais luminosidade é aplicada e menos da cor pode ser visto, o mesmo padrão é percebido para o escuro. Tendo assim variações sobre as cores conforme a luminosidade é alterada. Por fim o Croma define a intensidade da cor Matiz aplicada naquela direção (MUNSELL, 1929, p. 9-13).

Figura 12 - Representação tridemensional das cores de Munsell



Fonte: Munsell (1929, p. 1).

Seguindo a teoria apresentada por Munsell (1929), a imagem é alterada para o sistema de cores L*a*b*, sistema que tem seu desenvolvimento baseado na teoria, tendo então `L` definindo a luminância e `a` e `b` a crominância dos pixels da imagem. Como não interessa para o contexto da correlação a luminosidade externa que o pixel está recebendo, o canal `L` é então desconsiderado. Sendo então a fórmula da Figura 11 adaptada para considerar a distância euclidiana entre as crominâncias `a` e `b`. Podendo assim ser feita a correlação entre o pixel da primeira imagem e cada pixel da linha epipolar (FERNANDES, 2017c).

Para correlacionar o pixel da primeira imagem com um pixel da linha epipolar é analisado o resultado da fórmula. O menor resultado do cálculo de SSD adaptado de todos os pixels comparados na linha epipolar, é considerado o pixel correlato (FERNANDES, 2017c). Por fim para validar a correlação dos pixels foi sugerido por Fernandes (2017d) gerar um mapa de disparidade entre os pixels correlatos.

Segundo OpenCV (2014, p. 1, tradução nossa) “a profundidade de um ponto em uma cena é inversamente proporcional a diferença em distância dos pontos correlacionados com o

centro das câmeras.” Essa afirmação surgiu com base na equação da Figura 13, onde x e x' representam os pixels correlacionados da primeira e segunda imagem respectivamente.

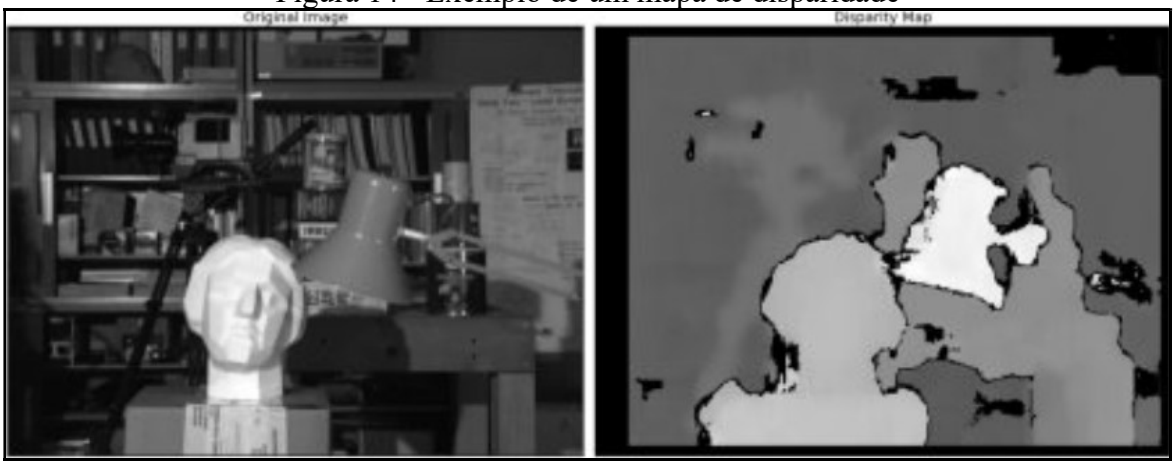
Figura 13 - Equação da distância entre os pontos correlacionados

$$\text{disparity} = x - x' = \frac{Bf}{Z}$$

Fonte: OpenCV(2014).

Para que seja possível gerar o mapa de disparidade, precisa ser gerada uma imagem em escala de cinza suportando o valor de um byte, entre 0 e 255. Este valor está diretamente relacionado a menor e maior distâncias encontradas entre todas as correlações, sendo assim necessário a normalização dos dados da distância para que eles possam representar um valor entre o canal suporte de um byte (FERNANDES, 2017d). Na Figura 14 é apresentado um exemplo de um mapa de disparidade.

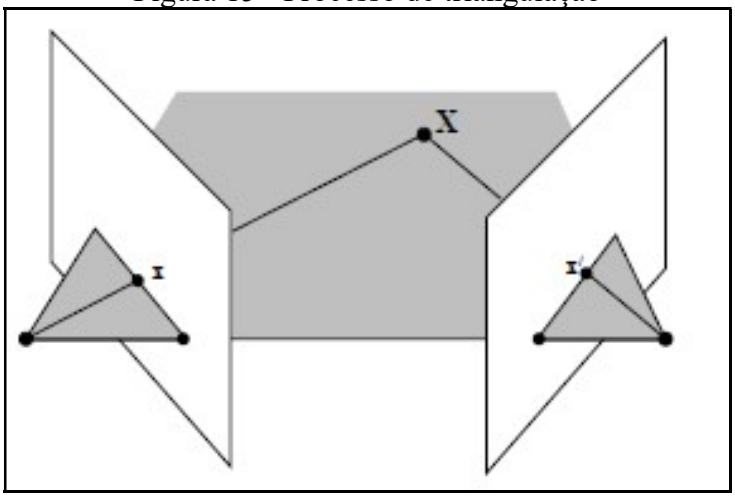
Figura 14 - Exemplo de um mapa de disparidade



Fonte: OpenCV (2014, p. 1).

A última etapa consiste em realizar a triangulação dos pixels correlacionados da etapa anterior. A triangulação consiste em aplicar um raio na cena atravessando os pontos correlatos, a fim de perceber a intersecção desses raios na cena 3D. Este processo pode ser visto na Figura 15 (FERNANDES, 2017c).

Figura 15 - Processo de triangulação

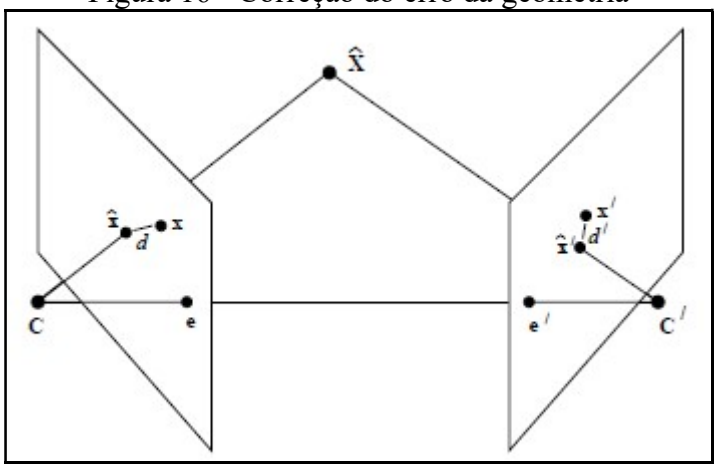


Fonte: Hartley e Zisseman (2004, p. 263).

Segundo Hartley e Zisseman (2004, p.310, tradução nossa) “Como há erros no processo de medição dos pontos `x` e `x′`, a projeção inversa do raio dos pontos é inclinada. Isso significa que não existirá um ponto `X` que satisfaça exatamente `x = PX, x′ = P'X`”. Sendo assim para realizar a etapa de triangulação torna-se necessário realizar uma etapa de correção do erro apresentado pela geometria.

Para corrigir este erro, podem ser aplicados alguns métodos lineares, iterativos e não iterativos. Dependendo do método deve-se tratar a taxa de erro quando projetar o raio, ou pode ser aplicado um algoritmo para correção da geometria que resultará em um novo ponto `x` e `x'`. Esses pontos são então utilizados para realizar a triangulação com um algoritmo de triangulação linear (FERNANDES, 2017c). A Figura 16 representa o algoritmo de correção da geometria para aplicação da retroprojeção do raio.

Figura 16 - Correção do erro da geometria



Fonte: Hartley e Zisseman (2004, p. 314).

Conforme sugerido por Fernandes (2017c) a solução que deve ser implementada é a correção da geometria para então ser feita a retroprojeção do raio. O algoritmo de correção da geometria tem por objetivo segundo Fernandes (2017c) “Dado `x` e `x′` e `F`, calcular a

correspondência correta que minimiza o erro geométrico.” Na Figura 17 é apresenta a fórmula que define o processo de correção da geometria.

Figura 17 - Fórmula de definição da correção da geometria

$$\min_{\mathbf{X}} C = d(\mathrm{x}, \hat{\mathrm{x}})^2 + d(\mathrm{x}', \hat{\mathrm{x}}')^2$$
$$= \min_{t} C = d\big(\mathrm{x}, \mathrm{l}(t)\big)^2 + d\big(\mathrm{x}', \mathrm{l}'(t)\big)^2$$

Fonte: Fernandes (2017c).

Conforme apresentado por Fernandes (2017c) o algoritmo consiste em um processo de 10 etapas que são apresentadas abaixo:

a) usar a translação T e T' para transladar os pontos $x=(x,\ y,\ 1)^T$ e $x'=(x',\ y',\ 1)^T$ para a origem;
b) substituir F atual por $T'^{-T}FT^{-1}$;
c) calcular os epipolos $e=(x_e,\ y_e,\ w_e)^T$ e $e'=(x_e{}',y_e{}',w_e{}')^T$ tal que $e'^TF=0$ e $Fe=0$. Garanta que $x_e^2 + y_e^2=1$ e $x_e{}'^2 + y_e{}'^2=1$;
d) monte as matrizes de rotação R e R' que resultem em $Re=(1,\ 0,w_e)^T$ e $R'e'=(1,\ 0,\ w_e{}')^T$;
e) substitua F atual por $R'^{-T}FR^{-1}$;
f) defina $f=w_e,\ f'=w_e{}',\ a=F_{2,2},\ b=F_{2,3},\ c=F_{3,2},\ d=F_{3,3}$;
g) forme o polinômio g(t) e encontre as 6 raízes
$g(t)=t\ ((at + b)^2 + f'^2\ (ct + d)^2)^2 - (ad - bc)(1 + f^2 + t^2)^2(at + b)(ct + d)$;
h) selecione o $\min(t)$ como sendo o valor de t resulta no menor custo $s(t)$ avaliado na parte real de cada uma das raízes de $g(t)$, na Figura 18 e no valor assimptótico do custo geométrico para $t=\text{infinito}$, na Figura 19;
i) defina as linhas epipolares $l=(\min(t)\ f,\ 1,\ -\min(t))^T$ e $l'=F\ (0,\ \min(t),\ 1)^T$ e encontre os novos x e x' como os pontos mais próximos da origem e sobre as linhas;
j) transfira os pontos encontrados de volta para a situação original substituindo o novo x por $T^{-1}R^{-1}x$ e o novo x' por $T'^{-1}R'^{-1}x'$;
k) o ponto 3D X finalmente é obtido por meio de um algoritmo de triangulação linear.

Figura 18 - Fórmula da distância quadrada total

$$s(t) = \frac{t^2}{1 + f^2 t^2} + \frac{(ct + d)^2}{(at + b)^2 + f'^2 (ct + d)^2}.$$

Fonte: Hartley e Zisseman (2004, p. 314).

Figura 19 - Valor assimptótico

$$\frac{1}{f^2} + \frac{c^2}{\left(a^2 + f'^2 c^2\right)^2}$$

Fonte: Fernandes (2017c).

2.3 TRABALHOS CORRELATOS

São apresentados três trabalhos correlatos que possuem características semelhantes a proposta deste trabalho. A seção 2.3.1 descreve o artigo de Li, Straub e Prautzsch (2004) que a partir da técnica de reconstrução por triangulação ativa e Geometria Epipolar, reconstruíram em um ambiente 3D o busto de Beethoven e um molde de uma caveira. A seção 2.3.2 apresenta a dissertação de mestrado de Zaaijer (2013), tendo em vista que o trabalho tem o foco de geração de relevos em tempo real, utilizando técnicas de processamento na unidade gráfica (Graphics Processing Unit, GPU), melhorando assim a performance com cálculos matemáticos complexos. Por fim, na seção 2.3.3 é apresentado um artigo (MANCINI et al, 2013) de um sistema capaz de gerar uma nuvem de pontos a partir de imagens tiradas por drones.

2.3.1 FAST SUBPIXEL ACCURATE RECONSTRUCTION USING COLOR STRUCTURED LIGHT

O artigo de Li, Straub e Prautzsch (2004, p. 1) apresenta o método de reconstrução por triangulação ativa, utilizando apenas uma imagem como entrada de dados. O trabalho propõe-se a utilizar um projetor como emissor da luz estruturada sobre o modelo e uma câmera como receptor da imagem para realizar a triangulação. O equipamento montado para realizar a captura é demonstrado na Figura 20.

Figura 20 - Projetor e câmera para realizar a triangulação óptica

Fonte: Li, Straub e Prautzsch (2004, p. 1).

A arquitetura do trabalho torna necessária a calibração entre o projetor e a câmera, sendo assim possível realizar a triangulação usando a lei dos senos, que leva como base o ângulo e a posição física entre os equipamentos. Após o processo de aquisição, as imagens captadas são fundidas em uma nuvem de pontos. Por último, é criada uma malha a partir da nuvem de pontos a fim de visualizar as propriedades do material (LI; STRAUB; PRAUTZSCH, 2004, p. 2).

Para criar o padrão de luz que será enviado pelo projetor foi utilizado o conceito de sequência cíclica de De Bruijn, que consiste em ter uma sequência de combinações a partir de um conjunto de números. O conjunto de números utilizados foi uma combinação baseada na expressão da Figura 21. Na detecção das bordas, entre as linhas das cores, surgem alguns problemas comuns no campo da VC como, por exemplo: interferência da luz do ambiente, textura do objeto e baixa densidade do pixel do projetor. Para minimizar o impacto destes problemas foram realizados três processos na imagem (LI; STRAUB; PRAUTZSCH, 2004, p. 2):

a) trata o fenômeno de *crosstalk* na cor, que acontece quando a luz do ambiente interfere na cor enviada pelo emissor, fazendo com que a cor utilizada como referência seja diferente da luz percebida. Para tratar isso, é definido que a luz refletida depende linearmente da luz emitida, para cada canal de cor enviado;
b) são removidas as áreas pretas da imagem onde a intensidade da luz é inferior a um limite definido pelo usuário;
c) um processo final de *smoothing* pode ser aplicado à imagem, se a imagem processada ainda conter muitos ruídos. É assumido que a imagem contém um ruído gaussiano e para corrigir isso é aplicado um filtro de proximidade *low-pass*.

Figura 21 - Expressão para caracterizar o conjunto da sequência de De Bruijn

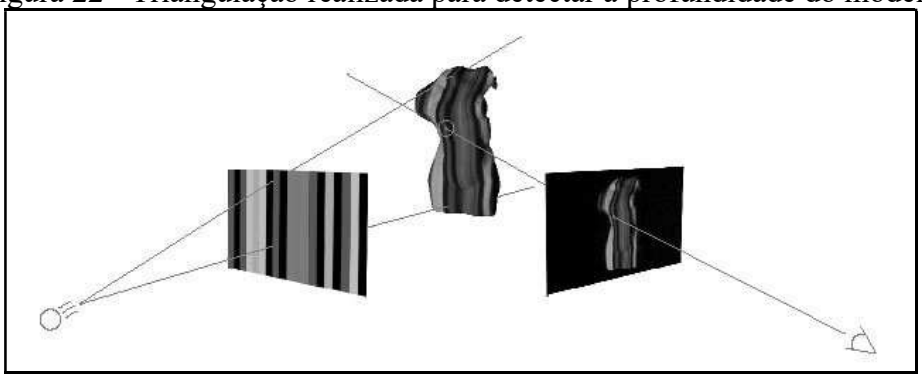
$$\mathbf{p}_i = (p_i^r, p_i^g, p_i^b) \in \{0,1\}^3$$

Fonte: Li, Straub e Prautzsch (2004, p. 2).

Para realizar o reconhecimento das linhas das cores considerasse que uma linha é sempre da mesma cor, assim fica necessário apenas o reconhecimento de uma dimensão nas linhas. Após realizar o procedimento de reconhecimento das cores é executada a identificação das bordas, a fim de identificar as arestas projetadas com as observadas na imagem. Este método é baseado num algoritmo de programação dinâmica. Ao final do processo é obtida uma matriz de custo do algoritmo, tendo assim a rotulação das imagens projetadas com a imagem obtida (LI; STRAUB; PRAUTZSCH, 2004, p. 3).

Segundo Li, Straub e Prautzsch (2004, p. 4) após realizar a rotulação das bordas de cada pixel da imagem capturada, deve se calcular a profundidade das bordas. Conforme Figura 22, o processo consiste em enviar um raio para cada pixel na imagem obtida pela câmera, verificando a intersecção gerada com a borda capturada no projetor. Neste caso o cálculo de profundidade é feito utilizando um algoritmo de Ray-Plane Intersection.

Figura 22 - Triangulação realizada para detectar a profundidade do modelo



Fonte: Li, Straub e Prautzsch (2004, p. 4).

2.3.2 GPU BASED GENERATION AND REAL-TIME RENDERING OF SEMI-PROCEDURAL TERRAIN USING FEATURES

Na dissertação de Zaaijer (2013, p. 3) são apresentados três métodos para a renderização de terrenos: métodos procedurais, métodos de simulação e métodos por esboços. O trabalho se propõe a gerar terrenos de larga proporção, mas com relevos simples e com um fácil controle do layout, para que seja possível manter um alto nível de performance foi usada a GPU.

A solução apresentada foi desenvolvida usando OpenGL para renderização do modelo e OpenCL para comunicação com a GPU. Segundo Zaaijer (2013, p. 6, tradução nossa), “a interoperabilidade entre essas duas APIs é comumente utilizada e facilitada.”

Para gerar o terreno é apresentado ao usuário uma ferramenta de desenho intuitiva tomando como base um sistema de camadas, onde cada camada representa um tipo de terreno, como montanha, rios, entre outros. O usuário pode especificar onde esse terreno será

renderizado usando formas. Podem ser dadas várias configurações para as formas, como por exemplo a altura da montanha, ângulo da superfície e rugosidade (ZAAIJER, 2013, p. 5).

Como o objetivo do trabalho é gerar malhas que possibilitam uma visão a distância e também possa ser utilizado o zoom para ter uma visão mais próxima dos detalhes do terreno, uma grande quantidade de dados deve ser armazenada. Com base nas limitações de renderização e armazenamento, gerar e apresentar todos os detalhes do terreno ao mesmo tempo não seria possível, pois se tiver um terreno de 4 km por 4 km e permitindo um detalhe de 25 cm de altura armazenando apenas usando uma variável do tipo float (4 bytes) para a altura seria necessário ter perto de 1 GB de dados para garantir a qualidade. Com toda essa quantidade de dados faz-se necessário o uso da GPU para se obter resultados em tempo real, devido a sua alta capacidade de processamento paralelo (ZAAIJER, 2013, p. 7).

Segundo Zaaijer (2013, p. 32), aplicações em tempo real deveriam trabalhar em média de 30 a 60 frames por segundo. Um dos problemas apresentados por ele foi a manipulação dos dados, pois como o OpenCL e o OpenGL usam os mesmos dados para renderização, eles não podem trabalhar em paralelo. Assim, para conseguir manter um bom frame rate, o kernel do OpenCL deve usar os dados o menor tempo possível.

2.3.3 USIGN UNMANNED AERIAL VEHICLES (UAV) FOR HIGH-RESOLUTION RECONSTRUCTION OF TOPOGRAPHY: THE SRTUCTURE FROM MOTION APPROACH ON COSTAL ENVIRONMENTS

A disponibilidade de modelos de superfícies digitais (Digital Surface Models, DSM) a partir de imagens aéreas, tiradas por drones (Veículos Aéreos Nâo Tipulados - VANTS) tem exigido alto nível de confiabilidade da topografia capturada. Com base na DSM capturada podem ser realizadas simulações de eventos naturais, levando em consideração as informações fornecidas pelas imagens. Um exemplo da importância das informações topográficas capturadas são os modelos de DSM obtidos a partir das costas das praias onde a presença de dunas aumentam a proteção natural afetando diretamente o resultado das simulações (MANCINI et al, 2013, p. 2).

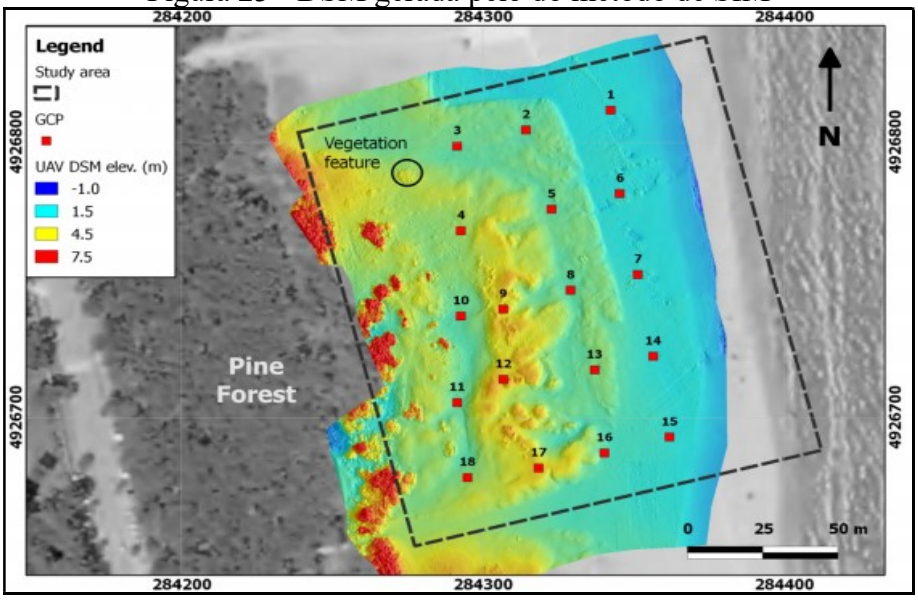
Apesar de existirem vários métodos para geração de nuvem de pontos e sucessivamente DSM de alta qualidade, são observados vários problemas que dificultam essa criação. Existem alguns métodos aplicados para realizar essa tarefa, dentre deles os mais utilizados são: Global Navigation Satellite Systems (GNSS), Terrestrial Laser Scanning (TLS), Light Detection And Ranging (LIDAR) e Structure from Motion (SfM). O método GNSS é rápido e preciso, mas possui um número limitado de pontos que pode gerar. O TLS

também possui uma precisão alta, mas necessita muito tempo para processar devido a anisotropia das dunas. A LIDAR não apresenta boa precisão na verticalidade comparada ao GNSS, TLS e o SfM (MANCINI et al, 2013, p. 2).

Levando em consideração os fatos apresentados, segundo Mancini et al (2013), foi escolhido o método SfM utilizando drones, por sua alta adesão na visão computacional. O objetivo do SfM é reconstruir uma cena 3D a partir de um conjunto de imagens de uma cena estática comparando a imagem com as imagens de referência tiradas pelo drone.

O software utilizado para gerar as imagens com o método de SfM foi o PhotoScan. Ele é capaz de gerar uma DSM a partir de no mínimo duas imagens do mesmo ponto. Na Figura 23 é apresentado um exemplo de saída gerada, tendo a altura com base no nível do mar. Após a finalização, os resultados foram comparados com os métodos de GNSS e TLS, para atestar e comparar com os métodos mais tradicionais. O processo de reconstrução divide-se em três etapas: a primeira é o alinhamento das imagens cadastradas. A partir de pontos de referência das imagens fornecidas e posição da câmera, o algoritmo fará a calibração dos elementos para identificação dos mesmos. O segundo passo é o processo denominado pixel-base dense stereo reconstruction no qual será gerada a nuvem de pontos da imagem e uma malha contendo as informações do relevo. Por último, uma textura é aplicada à malha, sendo que todo esse processo fica limitado apenas a quantidade de RAM disponível no computador (MANCINI et al, 2013, p. 8).

Figura 23 - DSM gerada pelo do método de SfM



Fonte Mancini et al (2013, p. 10).

Em conclusão, a técnica de SfM apresentada pelo artigo, quando comparada com técnicas que utilizam pontos de referência de geolocalização ou lasers para realizar a

reconstrução do relevo, comprovaram que a técnica de SfM pode ser utilizada e possui um tempo de processamento menor do que os outros dois, mantendo ainda uma alta precisão na verticalidade de dunas (MANCINI et al, 2013, p. 16).

3 DESENVOLVIMENTO DA BIBLIOTECA

Neste capítulo são demonstradas as etapas do desenvolvimento da biblioteca. Na seção 3.1 são apresentados os requisitos da biblioteca. A seção 3.2 descreve as especificações. A seção 3.3 apresenta as etapas detalhadas da implementação. Por fim, a seção 3.4 demonstra os resultados dos testes.

3.1 REQUISITOS

Os Requisitos Funcionais (RF) e os Requisitos Não Funcionais (RNF) da biblioteca de reconstrução de relevos são:

a) realizar a calibração da câmera e do projetor para a remoção da distorção (RF);
b) capturar imagens em tempo real da cena que será utilizada para realizar o reconhecimento do relevo a partir de uma WebCam (RF);
c) definir um padrão de imagem que será projetado permitindo a reconstrução 3D por meio da triangulação ativa (RF);
d) realizar a intersecção das linhas epipolares com a janela do padrão que está sendo projetado no ambiente (RF);
e) realizar a correlação do pixel da câmera com o correlato na imagem do projetor (RF);
f) gerar o mapa de disparidade para validar a correlação dos pixels (RF);
g) ser desenvolvido em C++ (RNF);
h) utilizar a biblioteca Mathfu para cálculos matemáticos (RNF);
i) utilizar a API CUDA para desenvolvimento na GPU (RNF);
j) ser desenvolvida com suporte a multiplataforma de arquitetura e sistema operacional (RNF);
k) utilizar o ambiente de desenvolvimento Visual Studio 2015 (RNF);
l) utilizar a ferramenta draw.io para modelagem dos diagramas (RNF);
m) ser desenvolvida utilizando a biblioteca OpenCV (RNF).

3.2 ESPECIFICAÇÃO

Nesta seção encontra-se uma descrição da estrutura da biblioteca, como o diagrama de classes e o diagrama de sequencias da etapa de calibração estéreo e da etapa de correlação entre os pixels.

3.2.1 DIAGRAMA DE CLASSES

Na Figura 24 é apresentado o diagrama de classes da biblioteca, onde pode ser visto como as classes estão estruturadas e suas ligações. Nesta seção são descritas as classes que compõem a estrutura base da biblioteca.

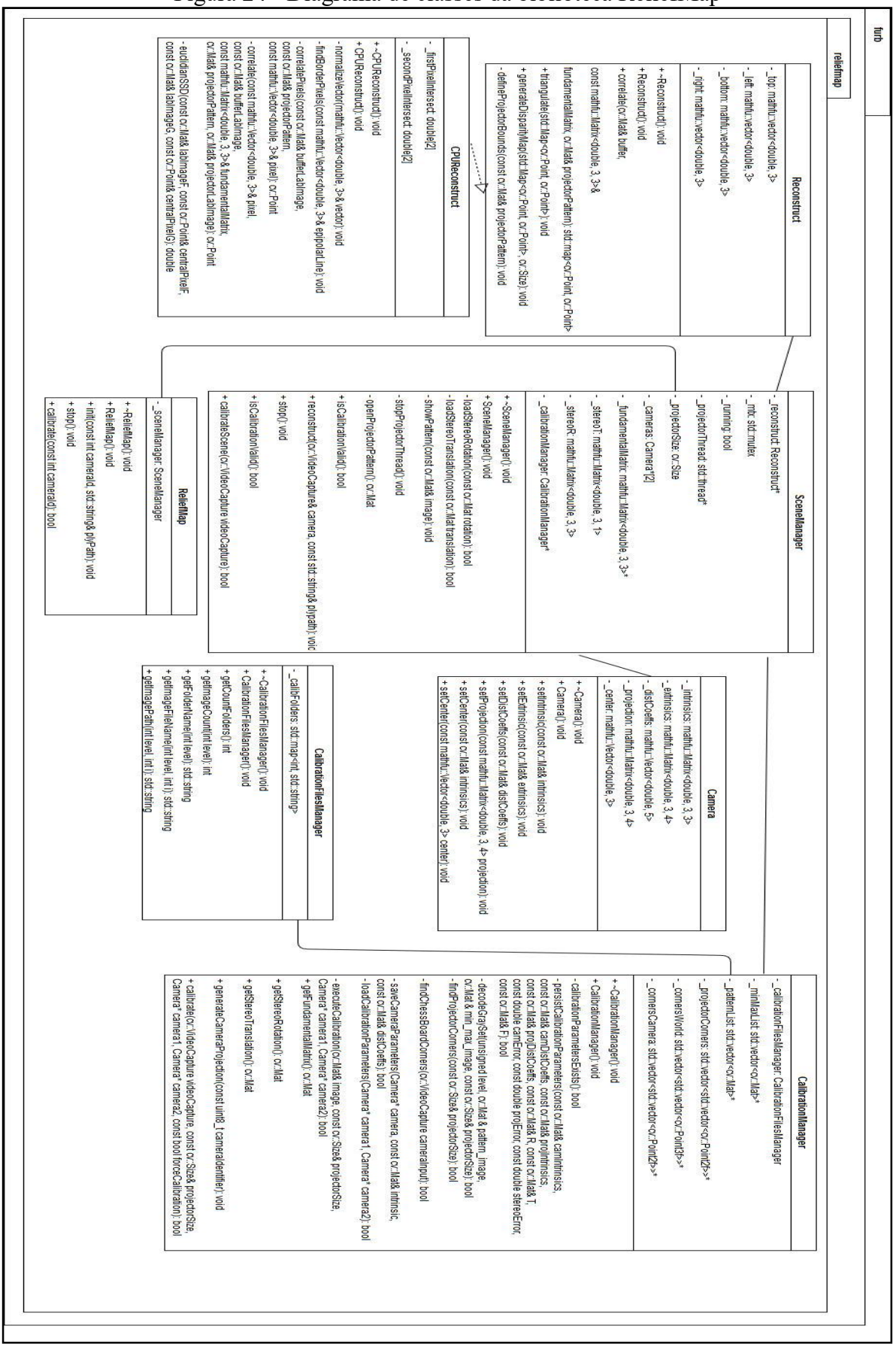
A classe `ReliefMap` é a classe de inicialização da biblioteca, nela estão disponíveis os métodos `calibrate` que é responsável por realizar o processo de calibração do ambiente estéreo. Para o método deve ser passado o identificador da câmera que será utilizada no processo. Após a calibração ter terminado, para começar o processo de reconstrução deve ser chamado o método `init`, também identificando a câmera que será utilizada e o local e o tipo de arquivo que será utilizado para gerar o modelo 3D. Caso queria pausar o processo de reconstrução por algum motivo, a classe fornece um método denominado `stop`, que pausa todos os processos que estão sendo utilizadas para reconstrução.

A classe `SceneManager` é responsável por controlar todo o fluxo entre as implementações de calibração e reconstrução. Fica a cargo desta classe armazenar os dados da calibração estéreo, como também armazenar os dados das câmeras e as referencias para as classes que realizam o processo de calibração e reconstrução. Na classe estão disponíveis os métodos de `init`, `stop`, `openProjectorThread`, `stopProjectorThread` e `isCalibrationValid`, que iniciam o processo de calibração e reconstrução, como também pausam estes processos. A classe de `Camera` serve como uma entidade de modelo para armazenar os dados da calibração das duas câmeras em cena.

As classes `CalibrationManager` e `CalibrationFilesManager` servem para controlar o fluxo de calibração da cena estéreo. A classe `CalibrationManager` é quem sabe como deve ser realizado o processo de calibração tanto da câmera, quanto do projetor em cena como uma câmera inversa, tendo também o processo de identificação do padrão de luz estruturado e realiza as chamadas para API OpenCV para a calibração estéreo. A classe `CalibrationFilesManager` tem o papel de fornecer os arquivos das imagens da calibração.

Por fim a classe de `Reconstruct` é uma classe abstrata que tem como métodos públicos os métodos de `correlate`, `reconstruct` e `generateDisparityMap`, que definem as funções mínimas que as classes herdeiras deverão implementar. Sendo está classe a responsável por permitir a futura otimização entre programação em CPU e GPGPU. O método `defineProjectorBounds` é implementado nesta classe por utilizar um algoritmo de baixa complexidade, definindo as equações das retas da borda do projetor. A classe `CPUReconstruct` é a implementação desta classe em CPU utilizando algoritmos lineares.

Figura 24 - Diagrama de classes da biblioteca ReliefMap

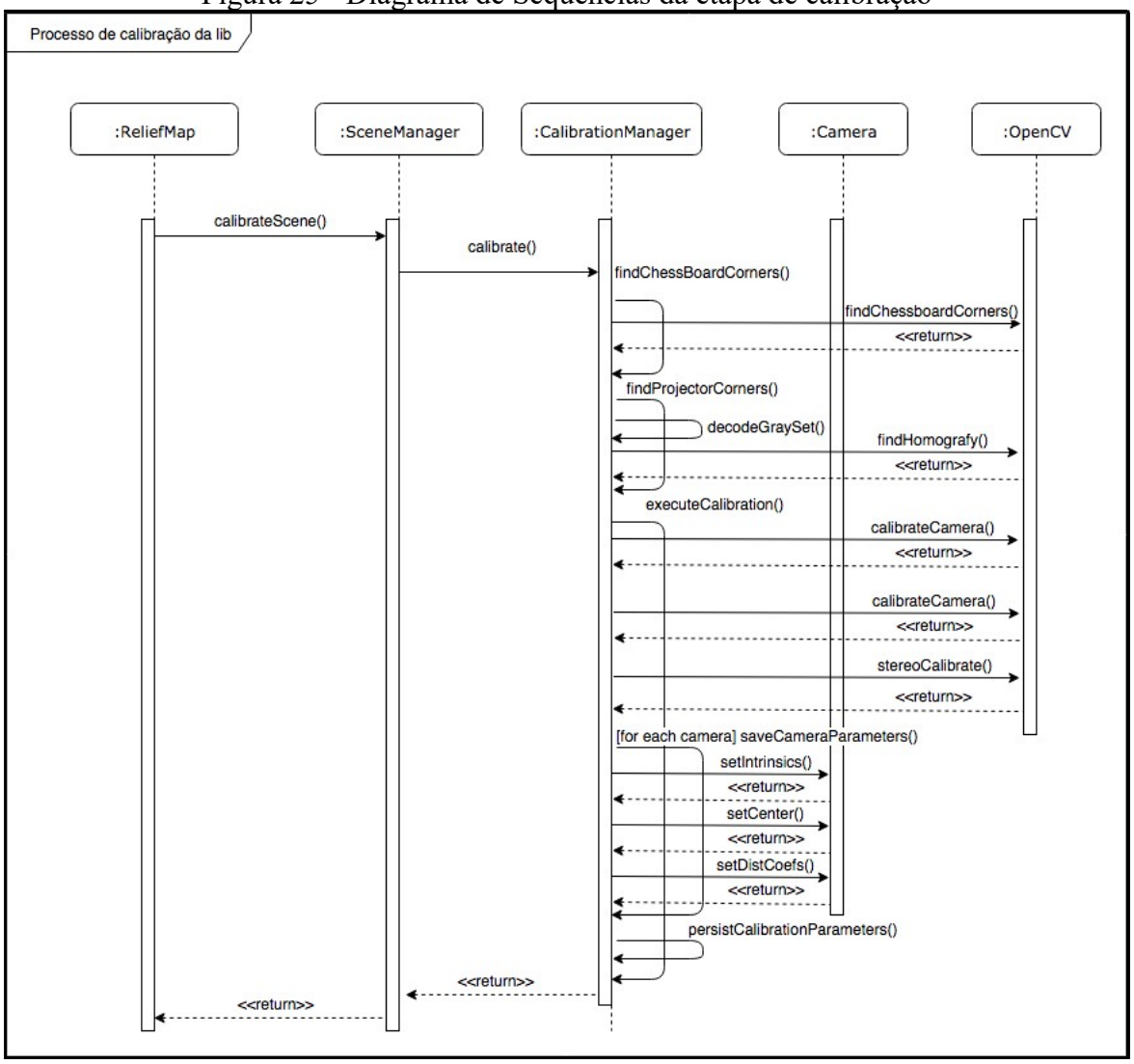


Fonte: elaborado pelo autor.

3.2.2 DIAGRAMA DE SEQUENCIAS DA ETAPA DE CALIBRAÇÂO

Na Figura 25 é apresentado o diagrama de sequencias consistindo na primeira etapa para a reconstrução 3D, a etapa de calibração da cena, sendo obrigatória e define a inicialização da biblioteca. O processo consiste em identificar o tabuleiro de xadrez nas imagens capturadas pelo software de Moreno e Taubin (2012), utilizando os processos implementados pela biblioteca OpenCV para identificar a distorção e parâmetros intrínsecos e extrínsecos da câmera. Após isto é realizado o reconhecimento da luz estruturada sobre as imagens em escala de cinza a fim de realizar a calibração do projetor. A próxima etapa de calibração consiste em realizar a calibração estéreo implementada pelo OpenCV, usando os valores obtidos nas calibrações da câmera e do projetor. Por fim todos os parâmetros são salvos nas instâncias das câmeras e persistidos em um arquivo para utilização futura.

Figura 25 - Diagrama de Sequencias da etapa de calibração



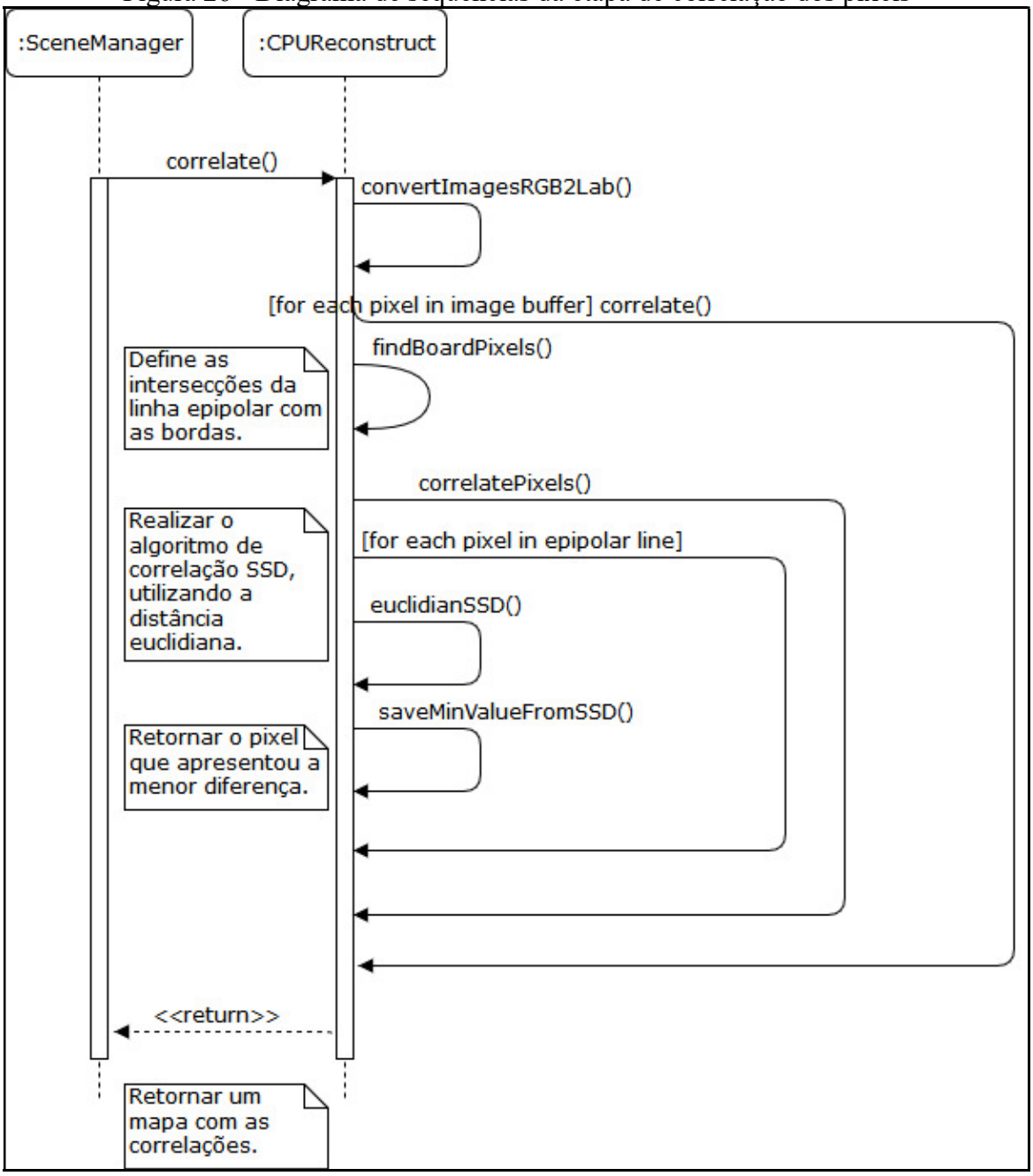
Fonte: elaborado pelo autor.

3.2.3 DIAGRAMA DE SEQUENCIAS PARA CORRELAÇÃO ENTRE PIXELS

Na Figura 26 é apresentado o diagrama de sequencias para correlação entre os pixels da primeira imagem com todos os pixels da linha epipolar na segunda imagem. A primeira etapa do algoritmo consiste em definir os pontos de intersecção entre as linhas epipolares e as bordas da imagem do projetor. Essa intersecção é feita com um cálculo de produto vetorial entre as equações de retas das bordas com a equação da reta da linha epipolar.

A segunda etapa do algoritmo começa a partir do método de `correlatePixels` após definir as intersecções com a borda. As etapas consistem em converter as duas imagens para os canais Lab. Após isto são calculados os valores da fórmula de SSD, a partir da distância euclidiana entre as cores nos canais a e b. O algoritmo consiste em realizar o somatório entre o *kernel* na primeira e segunda imagem. Tendo como resultado da correlação o menor valor entre os pixels da linha epipolar, sendo então considerado o correlato.

Figura 26 - Diagrama de sequencias da etapa de correlação dos pixels



Fonte: elaborado pelo autor.

3.3 IMPLEMENTAÇÃO

Nesta seção são descritas as ferramentas utilizadas para o desenvolvimento da biblioteca, bem como os algoritmos para calibração de câmera, calibração do projetor e calibração estéreo. Serão abordados também a etapa de correlação entre as linhas epipolares com as bordas da janela da imagem do projetor, bem como a etapa de correlação entre pixels utilizando seus valores de cores e etapas iniciais para a correção da geometria e triangulação. Por fim serão abordados os testes e os resultados obtidos.

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

No desenvolvimento da biblioteca foi utilizada a linguagem de programação C++ 11, o ambiente de desenvolvimento utilizado foi o Visual Studio Community 2015 14.25431.01 Update 3. Para o processamento paralelo na GPU foi utilizada a API CUDA 8.0.60.

A seguir, descrevem-se as fórmulas que foram utilizadas como base no desenvolvimento da aplicação, bem como as adaptações que foram realizadas com base nas pesquisas realizadas anteriormente.

3.3.1.1 ALGORITMO DE CALIBRAÇÃO DA CÂMERA

Conforme apresentado na seção 2.2.1, o algoritmo de calibração de uma câmera consiste em identificar características conhecidas num objeto no mundo real através da visão de uma câmera, sendo assim possível identificar a distorção causada nessas características. Foi utilizado durante o desenvolvimento uma imagem de um tabuleiro de xadrez para esta etapa, por conter tamanhos fixos entre os quadrados e ser representado num padrão de linhas e colunas. Para o processo de calibragem foi utilizada a imagem Anexo A do tabuleiro de xadrez fornecida pela API do OpenCV.

A biblioteca utiliza o método `findChessBoardCorners` implementado pelo OpenCV para identificar os quadrados do tabuleiro de xadrez. Este método necessita que sejam informados a quantidade de quadrados na horizontal e na vertical como também o tamanho (largura e altura) de cada quadrado. Para a imagem utilizada os valores do Quadro 1 foram aplicados.

Quadro 1 - Valores para detecção do tabuleiro de xadrez

CARACTERÍSTICA	VALOR
quantidade quadros horizontal	5
quantidade quadros vertical	6
largura e altura de cada quadrado	6

Fonte: elaborado pelo autor.

Segundo Zhang (2000, p. 8, tradução nossa) “é necessário tirar algumas imagens do modelo planar sobre diferentes orientações movendo o plano ou a câmera”. Na Figura 27 pode ser vista a imagem utilizada para o método de detecção dos quadrados do tabuleiro sobre a perspectiva de uma orientação. A posição dos quadrados em no mínimo 3 perspectivas distintas é armazenada para que seja possível realizar a calibração do projetor. No Quadro 2 é apresentado o código fonte de captura, sendo as adaptações da biblioteca a remoção da dependência da biblioteca de interface gráfica do software de Moreno e Taubin (2012).

Figura 27 - Imagem utilizada para identificação dos contornos do tabuleiro de xadrez

Fonte: elaborador pelo autor.

Quadro 2 – Código fonte para detecção dos quadrados do tabuleiro

```
cv::Size cornerCount(NUMB_CORNERS_HEIGHT, NUMB_CORNERS_WIDTH);
cv::Size2f cornerSize(CORNER_SIZE, CORNER_SIZE);

this->_cornersWorld->clear();
this->_cornersWorld->resize(count);
this->_cornersCamera->clear();
this->_cornersCamera->resize(count);

for (unsigned i = 0; i<count; i++){
cv::Mat grayImage = getImage(i, 2);
      if (grayImage.rows<1){
            continue;
      }

      cv::Mat small_img;

      if (imageScale>1){
            cv::resize(grayImage, small_img, cv::Size(grayImage.cols /
imageScale, grayImage.rows / imageScale));
      }else{
            grayImage.copyTo(small_img);
      }

      std::vector<cv::Point2f> & camCorners = this->_cornersCamera->at(i);
      std::vector<cv::Point3f> & worldCorners = this->_cornersWorld-
>at(i);

      if (cv::findChessboardCorners(small_img, cornerCount, camCorners,
            cv::CALIB_CB_ADAPTIVE_THRESH + cv::CALIB_CB_NORMALIZE_IMAGE))
{

            std::cout << " - corners: " << camCorners.size() << std::endl;
            for (int h = 0; h<cornerCount.height; h++) {
                  for (int w = 0; w<cornerCount.width; w++) {

      worldCorners.push_back(cv::Point3f(cornerSize.width * w,
cornerSize.height * h, 0.f));
                  }
            }
      } else {
            return false;
            std::cout << " - chessboard not found!" << std::endl;
      }

      for (std::vector<cv::Point2f>::iterator iter = camCorners.begin();
iter != camCorners.end(); iter++) {
            *iter = imageScale*(*iter);
      }

      if (camCorners.size()) {
            cv::cornerSubPix(grayImage, camCorners, cv::Size(11, 11),
cv::Size(-1, -1),
            cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30,
0.1));
      }
}
```

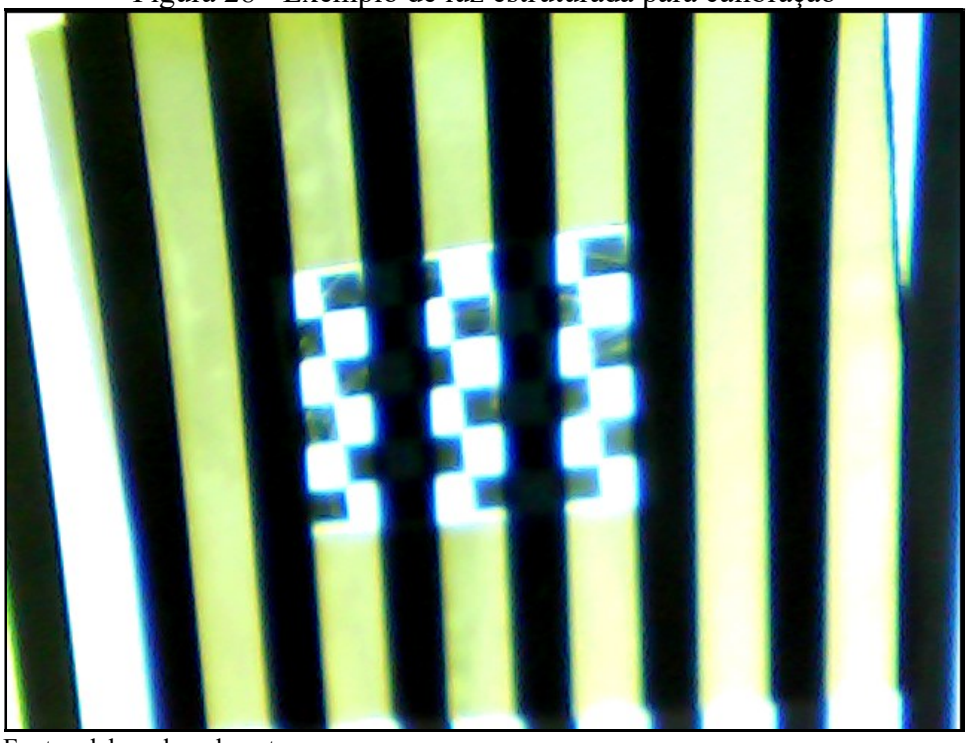
Fonte: Moreno e Taubin (2012) adaptado pelo autor.

3.3.1.2 ALGORITMO DE CALIBRAÇÃO DO PROJETOR

Segundo Moreno e Taubin (2012, p.4, tradução nossa) “Se o projetor fosse uma câmera, seria possível capturar imagens a partir do seu ponto de visão, para então procurar pelos cantos do tabuleiro de xadrez, e assim continuar como se fosse uma câmera.” Conforme apresentado na seção 2.2.1, não é possível capturar a imagem do projetor, sendo necessário realizar a etapa de calibração para a remoção da distorção. Moreno e Taubin (2012) apresentam uma técnica que consiste em utilizar luz estruturada em conjunto com a calibração da câmera, tendo assim como estipular a posição e visão do projetor sobre o tabuleiro de xadrez.

Segundo Moreno e Taubin (2012, p.4) o algoritmo pode ser dividido em três etapas, a primeira etapa consiste em decodificar a luz estruturada e associar cada pixel da câmera a uma linha e coluna da imagem projetada, ou marcado como incerto caso não seja possível identifica-lo. Um exemplo de luz estruturada emitida pelo algoritmo é vista na Figura 28. No Quadro 3 é apresentado o método de decodificação das imagens em escala de cinza. O algoritmo seleciona os pares de imagens com a luz estruturada na vertical e horizontal e carrega as posições dos objetos nessas imagens. Por fim, chama-se o método `decode_pattern` implementado por Moreno e Taubin (2012). sendo as adaptações da biblioteca a remoção da dependência da biblioteca de interface gráfica do software de Moreno e Taubin (2012).

Figura 28 - Exemplo de luz estruturada para calibração



Fonte: elaborado pelo autor.

Quadro 3 – Código fonte de decodificação das imagens em escala de cinza

```
patternImage = cv::Mat();
minMaxImage = cv::Mat();

const float b = 0.5;
const unsigned m = 5;

//estimate direct component
std::vector<cv::Mat> images;
int totalImages = this->_calibrationFilesManager.getImageCount(level);
int totalPatterns = totalImages / 2 - 1;
const int directLightCount = 4;
const int directLightOffset = 4;

if (totalPatterns<directLightCount + directLightOffset) {
      return false;
}

std::list<unsigned> directComponentImages;
for (unsigned i = 0; i<directLightCount; i++) {
      int index = totalImages - totalPatterns - directLightCount -
directLightOffset + i + 1;
      directComponentImages.push_front(index);
      directComponentImages.push_front(index + totalPatterns);
}

for each (unsigned i in directComponentImages){
      images.push_back(getImage(level, i));
}
cv::Mat directLight = sl::estimate_direct_light(images, b);

std::vector<std::string> imageNames;
unsigned levelCount = static_cast<unsigned>(this-
>_calibrationFilesManager.getImageCount(level));

for (unsigned i = 1; i<=levelCount; i++) {
      std::string filename = this-
>_calibrationFilesManager.getImagePath(level, i);

      imageNames.push_back(filename);
}

bool rv = sl::decode_pattern(imageNames, patternImage, minMaxImage, this-
>_projectorSize, sl::RobustDecode | sl::GrayPatternDecode, directLight,
m);
return rv;
```

Fonte: Moreno e Taubin (2012) adaptado pelo autor.

O segundo passo consiste em estimar uma homografia local entre cada canto do tabuleiro de xadrez na imagem da câmera. O processo consiste em analisar os cantos do tabuleiro encontrados pela câmera com o resultado da etapa anterior, a fim de gerar a homografia que é responsável por realizar a conversão deste ponto para as coordenadas do projetor. A homografia local foi utilizada para ser possível levar em consideração a distorção daquele ponto da imagem, melhorando assim a taxa de erro (MORENO; TAUBIN, 2012, p. 4). O algoritmo é apresentado no Quadro 4, sendo que algumas linhas de comentários e inicialização de valores foram removidas deste algoritmo para melhorar a visualização.

Quadro 4 - Algoritmo de definição da homografia local

```
227 for (std::vector<cv::Point2f>::const_iterator iter =
camCorners.cbegin(); iter != camCorners.cend(); iter++) {
228   const cv::Point2f & p = *iter;
229   cv::Point2f q;

231   //find an homography around p
233   unsigned WINDOW_SIZE = 60 / 2;
234   std::vector<cv::Point2f> imgPoints, projPoints;
235   if (p.x>WINDOW_SIZE && p.y>WINDOW_SIZE && p.x +
WINDOW_SIZE<patternImage.cols && p.y + WINDOW_SIZE<patternImage.rows) {
236         for (unsigned h = p.y - WINDOW_SIZE; h<p.y + WINDOW_SIZE;
h++) {
237               register const cv::Vec2f * row =
patternImage.ptr<cv::Vec2f>(h);
238               register const cv::Vec2b * minMaxRow =
minMaxImage.ptr<cv::Vec2b>(h);
240               for (unsigned w = p.x - WINDOW_SIZE; w<p.x +
WINDOW_SIZE; w++) {
241                     const cv::Vec2f & pattern = row[w];
242                     const cv::Vec2b & minMax = minMaxRow[w];
244                     if (sl::INVALID(pattern)) {
245                           continue;
246                     }
247                     if ((minMax[1] - minMax[0]) <
static_cast<int>(threshold)) {
248                           continue;
249                     }
251                     imgPoints.push_back(cv::Point2f(w, h));
252                     projPoints.push_back(cv::Point2f(patn));
255               }
256         }
257         cv::Mat H = cv::findHomography(imgPoints, projPoints,
CV_RANSAC);
259         cv::Point3d Q = cv::Point3d(cv::Mat(H *
cv::Mat(cv::Point3d(p.x, p.y, 1.0))));
260         q = cv::Point2f(Q.x / Q.z, Q.y / Q.z);
261   } else {
262         return false;
263   }
265   //save
266   projCorners.push_back(q);
267 }
```

Fonte: Moreno e Taubin (2012) adaptado pelo autor.

A última etapa é responsável por converter cada canto encontrado das coordenadas da câmera para as coordenadas do projetor aplicando a homografia encontrada para cada um deles. A equação utilizada para converter os cantos é apresentada na Figura 29 e também é implementada entre as linhas `257` a `260` do Quadro 4. Ao término dessas etapas tem-se os objetos mapeados na imagem do projetor possibilitando o processo de calibração do OpenCV, que define a matriz da câmera contendo os parâmetros intrínsecos e distorção radial (MORENO; TAUBIN, 2012, p. 4).

Figura 29 - Equação que define a homografia entre câmera e projetor

$$\hat{H} = \underset{H}{\operatorname{argmin}} \sum_{\forall p} ||q - Hp||^2$$

$$H \in \mathbb{R}^{3\times3}, \quad p = [x, y, 1]^T, \quad q = [col, row, 1]^T$$

$$\bar{q} = \hat{H} \cdot \bar{p}$$

Fonte: Moreno e Taubin (2012, p. 4) adaptado pelo autor.

3.3.1.3 ALGORITMO DE CALIBRAÇÂO ESTÉREO

O algoritmo de calibração estéreo consiste em realizar a calibração entre um par de câmeras observando um mesmo ambiente, conforme apresentado na seção 2.2.1. O projetor é considerado no algoritmo como sendo uma câmera inversa por não ser possível capturar a imagem por ele. Tendo finalizado as etapas das seções 3.3.1.1 e 3.3.1.2 torna-se possível realizar a etapa da calibração estéreo.

É apresentado então o código fonte do Quadro 5, onde algumas linhas com definição de teste e linhas que não alteram propriedades do algoritmo foram removidas para melhorar a visualização, sendo as adaptações da biblioteca a remoção da dependência da biblioteca de interface gráfica do software de Moreno e Taubin (2012). Na linha `452` e `466` o algoritmo realiza a etapa final de calibração da câmera e do projetor respectivamente, utilizando os valores dos cantos dos tabuleiros mapeados. Esta função retornará a matriz dos parâmetros intrínsecos e os coeficientes de distorção de cada câmera. Esses valores serão utilizados na função da calibração estéreo, que é vista na linha `476` do algoritmo, juntamente com os valores dos objetos do tabuleiro de xadrez. Essa função tem como retorno a rotação e translação necessária para levar um ponto da visão da câmera para a visão do projetor e também calcula o Matriz Fundamental apresentada na seção 2.2.2. Todas essas funções são implementadas pela biblioteca OpenCV.

Após a calibração ser finalizada todos os parâmetros obtidos nas três etapas de calibração serão salvos em um arquivo na pasta do projeto, sendo assim possível executar a aplicação posteriormente sem que seja necessário realizar as etapas de calibração. Por fim o ambiente estéreo estará calibrado e algoritmo segue com o processo de triangulação apresentado na seção 2.2.3.

Quadro 5 – Código fonte de calibração estéreo

```
424 unsigned count = static_cast<unsigned>(this-
>_calibrationFilesManager.getCountFolders());
425 cv::Mat camIntrinsics = cv::Mat(3, 3, CV_32FC1), camDistCoeffs;
426 cv::Mat projIntrinsics = cv::Mat(3, 3, CV_32FC1), projDistCoeffs;
427 int calFlags = 0 + cv::CALIB_FIX_K3;
428
429 std::vector<std::vector<cv::Point3f> > worldCornersActive;
430 std::vector<std::vector<cv::Point2f> > cameraCornersActive;
431 std::vector<std::vector<cv::Point2f> > projectorCornersActive;
432 worldCornersActive.reserve(count);
433 cameraCornersActive.reserve(count);
434 projectorCornersActive.reserve(count);
435 for (unsigned i = 0; i<count; i++) {
436   std::vector<cv::Point3f> const& worldCorners = this-
>_cornersWorld->at(i);
437   std::vector<cv::Point2f> const& camCorners = this-
>_cornersCamera->at(i);
438   std::vector<cv::Point2f> const& projCorners = this-
>_projectorCorners->at(i);
439   if (worldCorners.size() && camCorners.size() &&
projCorners.size()) {   //active set
440         worldCornersActive.push_back(worldCorners);
441         cameraCornersActive.push_back(camCorners);
442         projectorCornersActive.push_back(projCorners);
443   }
444 }
[…]
450 //calibrate the camera
451 std::vector<cv::Mat> camRvecs, camTvecs;
452 double camError = cv::calibrateCamera(worldCornersActive,
cameraCornersActive, image.size(), camIntrinsics, camDistCoeffs,
camRvecs, camTvecs, calFlags, cv::TermCriteria(cv::TermCriteria::COUNT
+ cv::TermCriteria::EPS, 50, DBL_EPSILON));
[…]
463 //calibrate the projector
464 std::vector<cv::Mat> projRvecs, projTvecs;
465 cv::Size projectorSize(1024, 768);
466 double projError = cv::calibrateCamera(worldCornersActive,
projectorCornersActive, projectorSize, projIntrinsics, projDistCoeffs,
projRvecs, projTvecs, calFlags,cv::TermCriteria(cv::TermCriteria::COUNT
+ cv::TermCriteria::EPS, 50, DBL_EPSILON));
[…]
475 cv::Mat R, T, E, F;
476 double stereoError = cv::stereoCalibrate(worldCornersActive,
cameraCornersActive, projectorCornersActive, camIntrinsics,
camDistCoeffs, projIntrinsics, projDistCoeffs, image.size() , R, T, E,
F, cv::CALIB_FIX_INTRINSIC + calFlags,
cv::TermCriteria(cv::TermCriteria::COUNT + cv::TermCriteria::EPS, 150,
DBL_EPSILON));
```

Fonte: Moreno e Taubin (2012) adaptado pelo autor.

3.3.1.4 ALGORITMO DE CORRELAÇÃO DA LINHA EPIPOLAR

Conforme apresentado na seção 2.2.3 o processo de correlação da equacação da reta das linhas epipolares com a sequência de pixels na imagem projetada, consiste em aplicar uma

sequência de cálculos do produto vetorial para encontrar as equações da reta das bordas da imagem como também os pontos de intersecção entre as bordas com a linha epipolar.

O código fonte do Quadro 6 apresenta a parte inicial do processo de correlação das linhas epipolares dos pixels da imagem da câmera. Onde tem-se o percorrimento de todos os pontos das linhas e colunas da matriz da imagem capturada. Esses pontos são transformados em um vector de 3 posições, tendo o `i`, `j` e `1` como valores do ponto da imagem, os dois primeiros representando a linha e coluna do ponto e `1` sendo a coordenada homogênea. Por fim o algoritmo faz a chamada para o método responsável por realizar a correlação da linha epipolar para um ponto e destrói a imagem de *debug* sendo exibida se a biblioteca estiver neste modo.

Quadro 6 - Código fonte de inicialização para correlação das linhas epipolares

```
void CPUReconstruct::correlate(cv::Mat& buffer, const
mathfu::Matrix<double, 3, 3>& fundamentalMatrix, cv::Mat&
projectorPattern) {
      this->defineProjectorBounds(projectorPattern);
      for (int i = 0, sizeR = buffer.rows; i < sizeR; ++i) {
            for (int j = 0, sizeC = buffer.cols; j < sizeC; ++j) {
                  mathfu::Vector<double, 3> pixel(i, j, 1);

                  this->correlate(pixel, buffer, fundamentalMatrix,
projectorPattern);

            }
      }

#ifdef DEBUG
      cv::destroyWindow("Test epipolar line");
#endif // DEBUG

}
```

Fonte: elaborado pelo autor.

No algoritmo do Quadro 7 temos a inicialização do processo de correlação, onde é feita a multiplicação da matriz fundamental com o ponto da imagem. Tendo como resultado um vector de 3 posições que representa a equação da reta desta linha. Com a equação é feita a chamada para o método `findBorderPixels` que aplica os produtos vetoriais sobre as bordas e a linha calculada.

Quadro 7 - Etapa inicial do método correlate

```
mathfu::Vector<double, 3> epipolarLine = fundamentalMatrix * pixel;
this->findBorderPixels(epipolarLine);
```

Fonte: elaborado pelo autor.

No Quadro 8 tem-se o código fonte que gera as intersecções entre a linha e as bordas. Após calcular o produto vetorial da linha e das bordas é necessário realizar a normalização do resultado para encontrar o ponto na imagem. O processo de normalização consiste em dividir

os valores `x` e `y` pelo valor `z` obtidos e atribuir `1` ao valor de `z`. Este processo é realizado no método `normalizeVector`.

Quadro 8 - Implementação do método findBoardPixels

```
void CPUReconstruct::findBorderPixels(const mathfu::Vector<double, 3>&
epipolarLine) {
// acha as intersecções das bordas com a linha epipolar
      mathfu::Vector<double, 3> intersectTop = mathfu::Vector<double,
3>::CrossProduct(epipolarLine, this->_top);
      this->normalizeVector(intersectTop);

      mathfu::Vector<double, 3> intersectRight = mathfu::Vector<double,
3>::CrossProduct(epipolarLine, this->_right);
      this->normalizeVector(intersectRight);

      mathfu::Vector<double, 3> intersectBottom =
mathfu::Vector<double, 3>::CrossProduct(epipolarLine, this->_bottom);
      this->normalizeVector(intersectBottom);

      mathfu::Vector<double, 3> intersectLeft = mathfu::Vector<double,
3>::CrossProduct(epipolarLine, this->_left);
      this->normalizeVector(intersectLeft);

      if (intersectBottom.x > intersectLeft.x && intersectBottom.x <
intersectRight.x) {
            this->defineIntersection(intersectBottom, this-
>_firstPixelIntersect);
      } else if (intersectBottom.x < intersectLeft.x) {
            this->defineIntersection(intersectLeft, this-
>_firstPixelIntersect);
      } else if (intersectBottom.x < intersectRight.x) {
            this->defineIntersection(intersectRight, this-
>_firstPixelIntersect);
      }

      if (intersectTop.x > intersectLeft.x && intersectTop.x <
intersectRight.x) {
            this->defineIntersection(intersectTop, this-
>_secondPixelIntersect);
      } else if (intersectTop.x < intersectLeft.x) {
            this->defineIntersection(intersectLeft, this-
>_secondPixelIntersect);
      } else if (intersectTop.x < intersectRight.x) {
            this->defineIntersection(intersectRight, this-
>_secondPixelIntersect);
      }
}
```

Fonte: elaborado pelo autor.

No fim do método do Quadro 8 são definas quais as intersecções mais próximas as bordas e as atribui como sendo os pontos para o algoritmo de Bresenham, sendo que foi utilizado o objeto `LineIterator` do OpenCV para realizar este processo.

A etapa de definição das equações das retas das bordas do padrão de projeção foi implementada fora do processo dos métodos apresentados, pois como o padrão não é alterado em tempo de execução tem seu tamanho fixo, podendo assim ser calculado apenas uma vez

para todos pontos da imagem. O Quadro 9 apresenta o código fonte que realiza a sequência de cálculos do produto vetorial entre os pixels das bordas da imagem.

Quadro 9 - Código fonte que define as equações da reta das bordas da projeção

```
inline void defineProjectorBounds(const cv::Mat& projectorPattern) {
      mathfu::Vector<double, 3> topLeft(0, 0, 1);
      mathfu::Vector<double, 3> topRight(projectorPattern.cols - 1, 0,
1);
      mathfu::Vector<double, 3> bottomLeft(0, projectorPattern.rows -
1, 1);
      mathfu::Vector<double, 3> bottomRight(projectorPattern.cols - 1,
projectorPattern.rows - 1, 1);

      this->_top = mathfu::Vector<double, 3>::CrossProduct(topLeft,
topRight);

      this->_right = mathfu::Vector<double, 3>::CrossProduct(topRight,
bottomRight);

      this->_bottom = mathfu::Vector<double,
3>::CrossProduct(bottomRight, bottomLeft);

      this->_left = mathfu::Vector<double,
3>::CrossProduct(bottomLeft, topLeft);
}
```

Fonte: elaborado pelo autor.

3.3.1.5 ALGORITMO DE CORRELAÇÃO DOS PIXELS

O algoritmo de correlação dos pixels consiste em alterar a fórmula de SSD, para levar em consideração a distância euclidiana entres as cores da imagem capturada e a projetada. Conforme apresentado na segunda etapa da seção 2.2.3, o código fonte leva em consideração a distância euclidiana das cores no sistema de cores L*a*b*, que segundo Fernandes (2017c) aprimoram a chances de correlação dos pixels.

No Quadro 10 é apresentado o código fonte que percorre todos os pixels encontrados na linha epipolar, utilizando a implementação `cv::LineIterator` da API do OpenCV. O processo realiza o cálculo SSD, somando as distâncias euclidianas dos canais de cores `a` e `b` do *kernel* sobre os pixels na primeira imagem da câmera e na segunda imagem do padrão estruturado. O menor valor que retorna do método `euclidianSSD` é considerado pixel correlato.

Quadro 10 – Código fonte de correlação de um pixel com os pixels da linha epipolar

```
cv::Point CPUReconstruct::correlatePixels(const cv::Mat&
bufferLabImage, const cv::Mat& projectorLabImage, const
mathfu::Vector<double, 3>& pixel) {

	cv::Point correlation;

	// Cria um iterador sobre todos os pontos da linha epipolar..
	cv::LineIterator it(projectorLabImage,
			cv::Point(this->_firstPixelIntersect[0], this-
>_firstPixelIntersect[1]),
			cv::Point(this->_secondPixelIntersect[0], this-
>_secondPixelIntersect[1]),
			8);

	double value = std::numeric_limits<double>::max();
	for (int i = 0; i < it.count; i++, ++it) {
		cv::Point pixelProj = it.pos();
		double ssd = this->euclidianSSD(bufferLabImage,
cv::Point(pixel.x, pixel.y), projectorLabImage, cv::Point(pixelProj.x,
pixelProj.y));

		if (ssd < value) {
			value = ssd;
			correlation.x = pixelProj.x;
			correlation.y = pixelProj.y;
		}
	}

	return correlation;
}
```

Fonte: elaborado pelo autor.

No Quadro 11 é apresentado o código fonte que realiza o cálculo SSD euclidiano dos canais `a` e `b` da imagem. O algoritmo consiste em percorrer os pixels da imagem, considerando o tamanho *kernel*, validando se a posição está contida dentro da imagem. Ou seja, caso o pixel central seja o `0, 0` o *kernel* não pode acessar as posições da linha `-1` nem as posições da coluna `-1`. Para realizar está validação o algoritmo verifica qual o valor mediano do *kernel* para definir quantos passos em todas as direções o algoritmo pode capturar os pixels, validando assim se o valor é maior que `0` e menor que a quantidade de pixels na linha e coluna. Os valores que definem o tamanho do *kernel* são definidos através das constantes `HEIGHT_MASK` e `WIDTH_MASK`, atualmente ambos valores estão definidos com o valor `3`.

O valor do *kernel* deve sempre representar um quadrado, contendo em seu interior um pixel central, sendo assim necessário um valor ímpar na altura e largura. Por fim, o algoritmo retorna o somatório das distâncias euclidianas entre os pixels das duas imagens.

Quadro 11 - Código fonte que calcula do algoritmo de SSD

```
double CPUReconstruct::euclidianSSD(const cv::Mat& labImageF, const
cv::Point& centralPixelF, const cv::Mat& labImageG, const cv::Point&
centralPixelG) {

	double sum = 0;

	// percorre a máscara nos píxels da imagem.
	int middleHeight = HEIGHT_MASK / 2;
	int middleWidth = WIDTH_MASK / 2;
	for (int i = 0; i < HEIGHT_MASK; ++i) {
		int footsHeigth = i - middleHeight;
		int newXF = centralPixelF.x + footsHeigth;
		int newXG = centralPixelG.x + footsHeigth;

		if (newXF >= 0 && newXF < labImageF.rows &&
			newXG >= 0 && newXG < labImageG.rows) {
			for (int j = 0; j < WIDTH_MASK; ++j) {
				int footsWidth = j - middleWidth;
				int newYF = centralPixelF.y + footsWidth;
				int newYG = centralPixelG.y + footsWidth;

				if (newYF >= 0 && newYF < labImageF.cols &&
					newYG >= 0 && newYG < labImageG.cols) {
					cv::Vec3b labPixelF =
labImageF.at<cv::Vec3b>(newXF, newYF);
					cv::Vec3b labPixelG =
labImageG.at<cv::Vec3b>(newXG, newYG);

					double a = labPixelF.val[1] -
labPixelG.val[1];
					double b = labPixelF.val[2] -
labPixelG.val[2];
					double euclidianDist = a * a + b * b;

					sum += euclidianDist * euclidianDist;
				}
			}
		}
	}

	return sum;
}
```

Fonte: elaborador pelo autor.

3.3.1.6 ALGORITMO PARA GERAÇÃO DO MAPA DE DISPARIDADE

Para ser possível validar a correlação dos pixels das duas imagens pode ser criado o mapa de disparidade entre as correlações. O mapa de disparidade consiste em compara as distâncias euclidianas entre as duas correlações, onde as maiores distâncias são mapeadas para o mais próximo do valor `255` e as menores distâncias para o valor `0`. Sendo esse mapeamento feito com base em uma normalização do maior valor para `255` e do menor valor para `0`, entre todas as distâncias calculadas.

Conforme implementação do método `generateDisparityMap` no Quadro 12, a primeira parte do código fonte consiste em verificar as distâncias euclidianas entre todas as

correlações, já sendo salvo o maior e o menor valor. Enquanto na segunda parte é feita a etapa de normalização, que tem como retorno um valor entre `0` e `1`, que equivale a proximidade do valor atual com o menor e maior valor. Este valor de normalização é então multiplicado por `255` e armazenado em um `int` para definir como valor do pixel na imagem em escala de cinza. Por fim a imagem em escala de cinza é salva na pasta raiz do projeto que está usando a biblioteca.

Quadro 12 - Código fonte de geração do mapa de disparidade

```
void
CPUReconstruct::generateDisparityMap(std::vector<std::tuple<cv::Point,
cv::Point>> correlations, cv::Size& imageSize) {
	double min = std::numeric_limits<double>::max();
	double max = 0;
	std::vector<std::tuple<cv::Point, double>> distances;
	for (int i = 0, size = correlations.size(); i < size; ++i) {
		std::tuple<cv::Point, cv::Point> correlation =
correlations[i];
		double  x = std::get<1>(correlation).x -
std::get<0>(correlation).x;
		double  y = std::get<1>(correlation).y -
std::get<0>(correlation).y;

		double distance = sqrt(x*x + y*y);
		distances.push_back(std::tuple<cv::Point,
double>(std::get<0>(correlation), distance));

		if (distance > max) {
			max = distance;
		}
		if (distance < min) {
			min = distance;
		}
	}

	cv::Mat img(imageSize, CV_8UC1, cv::Scalar(255));
	for (int i = 0, size = distances.size(); i < size; ++i) {
		std::tuple<cv::Point, int> distancy = distances[i];
		cv::Point pixel = std::get<0>(distancy);
		if (pixel.x < imageSize.height && pixel.y <
imageSize.width) {
			double normalized = (std::get<1>(distancy) - min) /
(max - min);
			uchar pixelValue = normalized * 255;
			img.at<uchar>(pixel.x, pixel.y) = pixelValue;
		}
	}
	cv::imwrite("disparityImage.jpg", img);
}
```

Fonte: elaborado pelo autor.

3.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Como o projeto trata-se de uma biblioteca para utilização em outras aplicações, não é disponibilizado nenhuma interface de comunicação com o usuário. A fim de demonstrar a utilização da biblioteca, foi desenvolvida uma aplicação que realiza as chamadas necessárias

para realizar os processos de calibração, inicialização e pausa da biblioteca. Conforme Quadro 13, a utilização consiste na criação de uma instância da classe `ReliefMap` da biblioteca e na chamada dos métodos `calibrate` e `init`, especificando qual câmera deseja-se usar para a calibração e reconstrução.

Caso o usuário queira parar por um tempo a execução da biblioteca pode ser acionado o método `stop` que pausará o processo assim que possível. No método `init` é possível também especificar o caminho e o nome onde o usuário deseja que a biblioteca salve o arquivo `.ply`, sendo esse, o arquivo da nuvem de pontos gerado pela reconstrução 3D final. Caso não seja especificado o caminho, a biblioteca salva-o na pasta raiz do projeto com o nome de `reconstruct.ply`.

Quadro 13 - Utilização da biblioteca

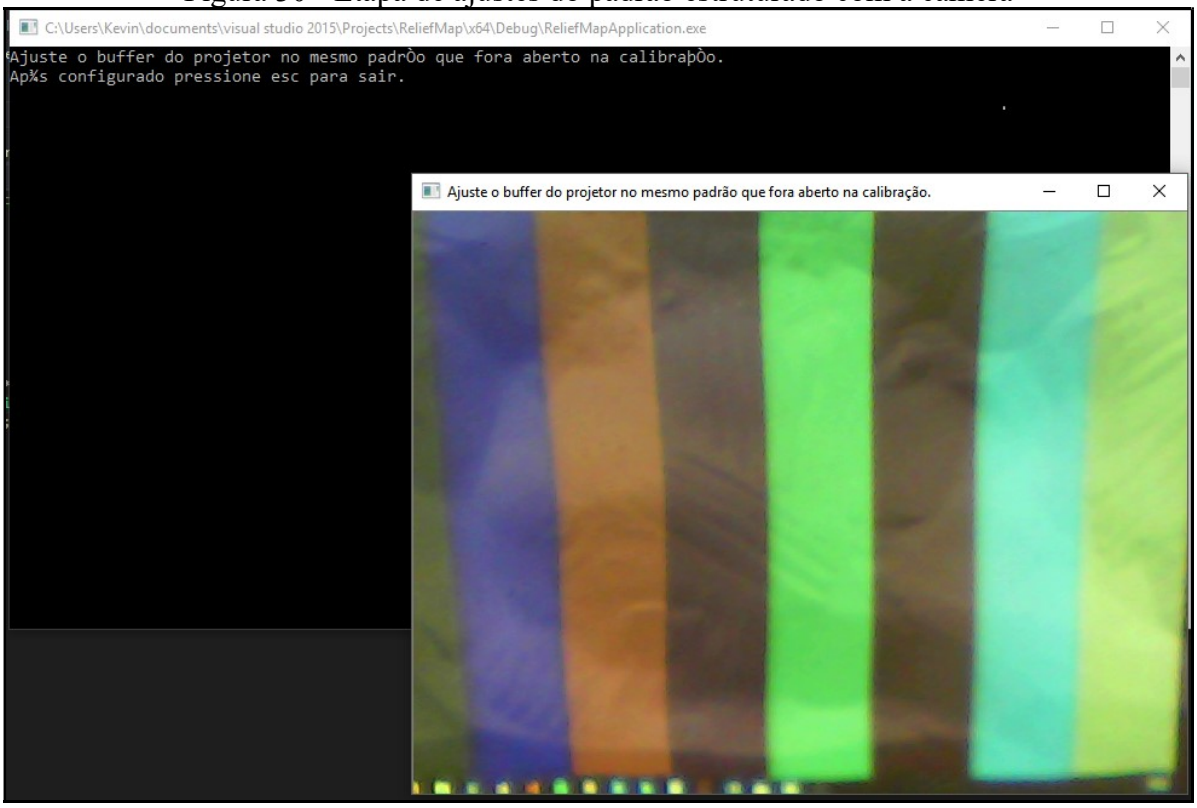
```
int main(int argc, char **argv) {
      furb::reliefmap::ReliefMap library;
      library.calibrate(0);
      library.init(0);
      library.stop();

      return 0;
}
```

Fonte: elaborado pelo autor.

Algumas dependências tornam-se necessárias para que a biblioteca consiga realizar todas as etapas. A primeira dependência encontra-se com a necessidade, de na pasta raiz da aplicação que usa a biblioteca, ter uma pasta chamada `calib`, contendo no mínimo outras três pastas com as imagens de calibração do ambiente estéreo capturadas pelo software desenvolvido por Moreno e Taubin (2012).

Após realizar a etapa de calibração, que consiste em realizar as etapas definidas na seções 3.3.1.1, 3.3.1.2 e 3.3.1.3, ou caso o processo já tenha sido feito e o arquivo de calibração exista, a biblioteca segue para próxima etapa. Em ambos os casos os dados das calibrações das câmeras e da calibração estéreo estarão carregados na classe `SceneManager`. Após este carregamento, será exibido pelo projetor o padrão de luz estruturada para que o usuário consiga calibrar e ajustar este padrão com a imagem que está sendo capturada pela câmera, conforme visto na Figura 30. Após este ajuste é necessário que seja pressionado a tecla `esc` na tela de visualização da imagem da câmera. Sendo executado em um *main loop* o processo de reconstrução a partir das imagens visualizadas.

Figura 30 - Etapa de ajustes do padrão estruturado com a câmera



Fonte: elaborado pelo autor.

3.4 ANÁLISE DOS RESULTADOS

A seguir são descritos os processos de testes realizados sobre a calibração da cena, a triangulação usando a Geometria Epipolar e a aplicação testes criados para validar as intersecções das linhas epipolares e correlação dos pixels. São apresentados também os resultados, as conclusões iniciais que podem ser tiradas a partir deste e alguns problemas identificados.

3.4.1 CALIBRAÇÃO DA CENA

A finalidade dos testes a seguir é verificar a integridade e o processo de calibração adaptado do trabalho de Moreno e Taubin (2012), conforme apresentado nas seções 2.2.1 e 3.3.1.1. Para realizar este teste foram capturadas as imagens usando o software disponibilizado por Moreno e Taubin (2012), o qual necessita de no mínimo 3 *sets* de imagens. Cada *set* contém 42 imagens com padrão de luz diferentes, sendo assim possível realizar a detecção da luz gerada pelo projetor na imagem e realizar a calibração tanto da câmera quanto do projetor.

A validação da calibração é feita pelo retorno do método de calibração da câmera e da calibração estéreo implementada pelo OpenCV, onde estes métodos retornam a taxa de erro

da calibração. Conforme visto na Tabela 1 após a realização dos testes, pode ser percebido que tanto o software de Moreno e Taubin (2012), quanto a adaptação da biblioteca apresentam taxas de erros distintas no mesmo *set* de imagens.

Tabela 1 - Comparação entre taxa de erro das execuções dos algoritmos de calibração

Taxa de erro	**Moreno e Taubin (2012)**	**Algoritmo adaptado pelo autor**
Calibração camera	0.403407	0.402455
Calibração projetor	0.766335	0.704214
Calibração estéreo	0.677827	0. 633276

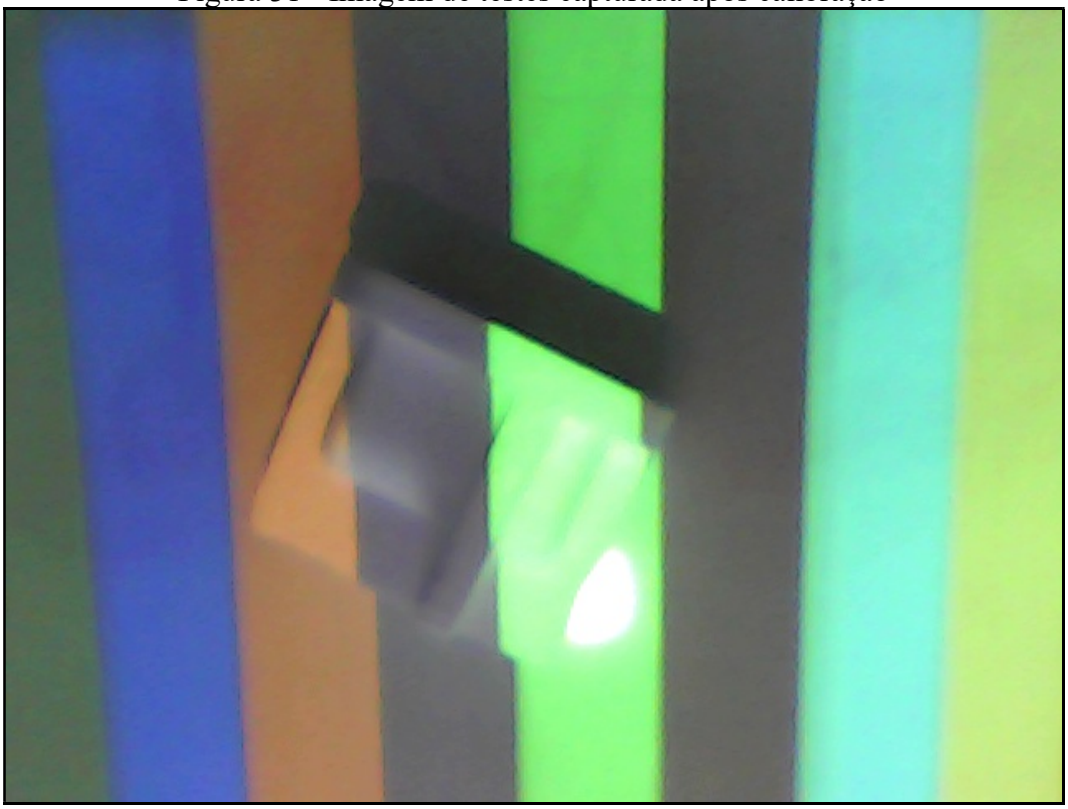
Fonte: elaborado pelo autor.

Mesmo tendo uma diferença entre as taxas de erro obtidas, segundo Moreno e Taubin (2012, p. 6, tradução nossa) “Obter uma taxa de erro mínima não garante a melhor precisão na reconstrução de um objeto arbitrário.” Sendo assim, como a taxa de erro do algoritmo da biblioteca é semelhante ao algoritmo original, o processo de calibração pode ser considerado válido. Tendo a vantagem de ter sido removida a dependência da biblioteca de construção de interfaces utilizada por Moreno e Taubin (2012).

3.4.2 INTERSECÇÃO DAS LINHAS EPIPOLARES

Conforme apresentado na primeira etapa da seção 2.2.3, é necessário realizar o processo de intersecção da linha epipolar, com as retas das bordas da segunda imagem. O algoritmo teve como entrada a imagem do padrão estruturado e a imagem da Figura 31, que consistia em uma imagem de testes captura após a calibração do ambiente estéreo.

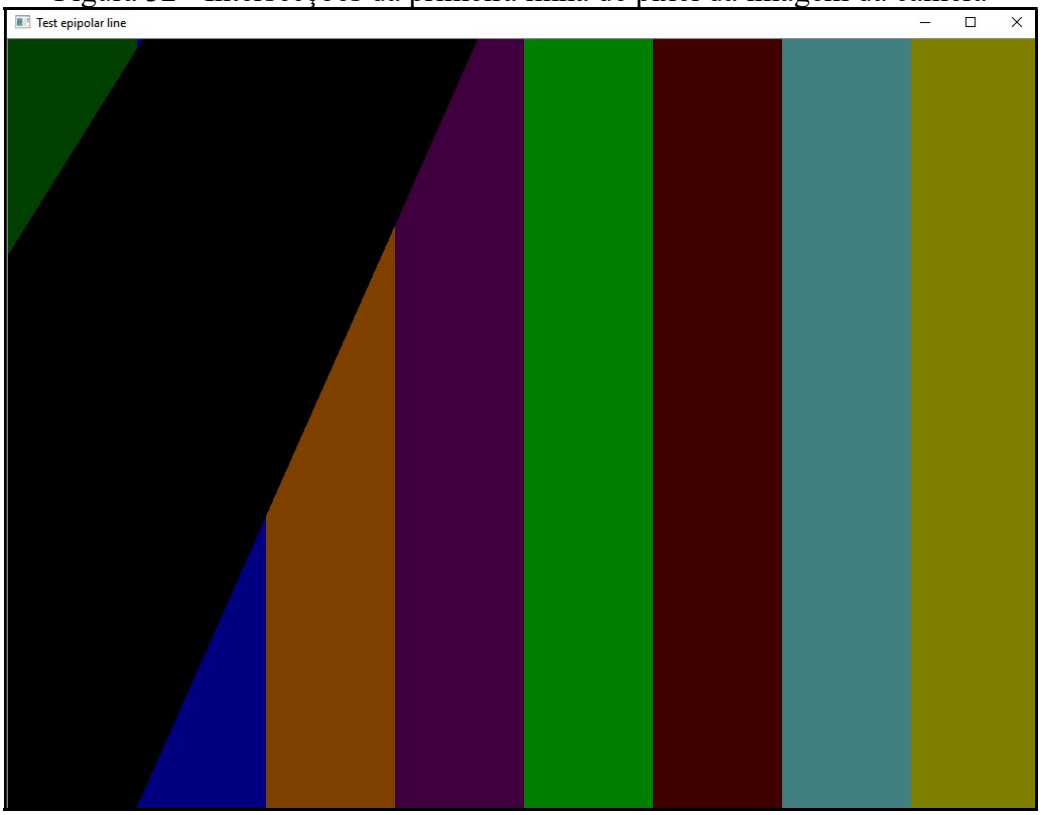
Figura 31 - Imagem de testes capturada após calibração



Fonte: elaborado pelo autor.

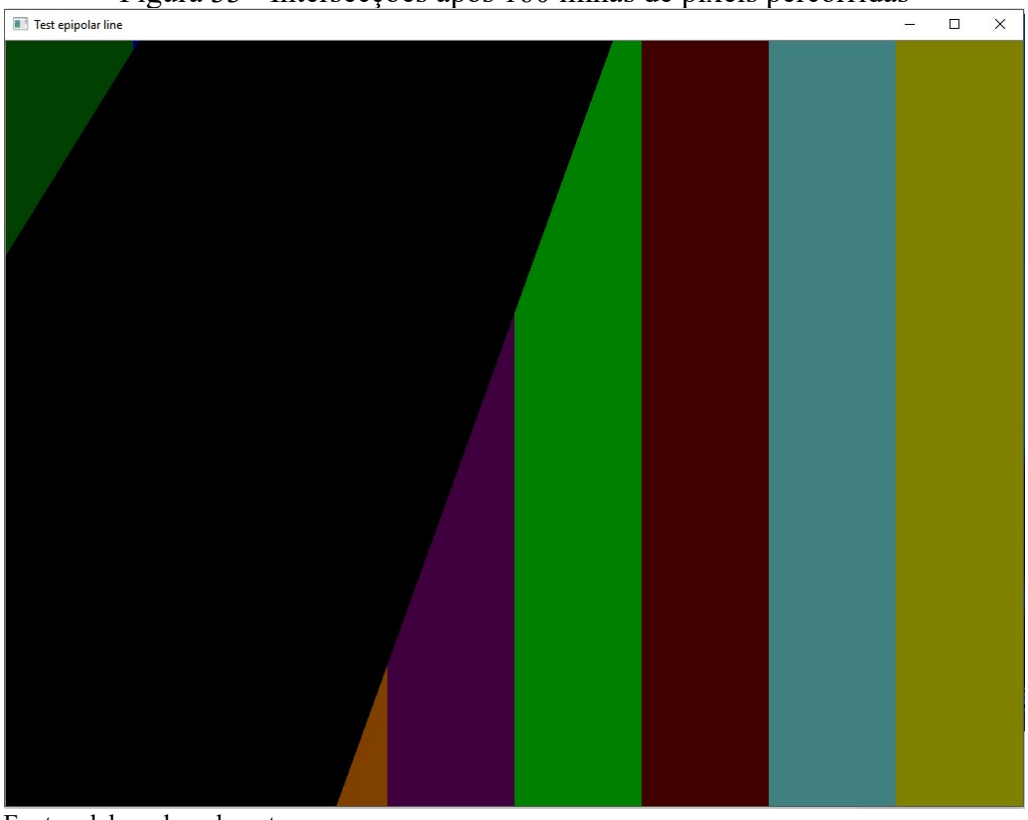
Durante o processo de intersecção entre os pixels da primeira imagem com as linhas epipolares do projetor foram obtidas as imagens Figura 32, Figura 33, Figura 34 e Figura 35. Estas imagens comprovam que as linhas epipolares estão navegando pela imagem em uma sequencia lógica, sendo que a leitura dos pixels da primeira imagem começa no pixel superior esquerdo seguido pelos pixels da sua linha.

Figura 32 - Intersecções da primeira linha de pixel da imagem da câmera



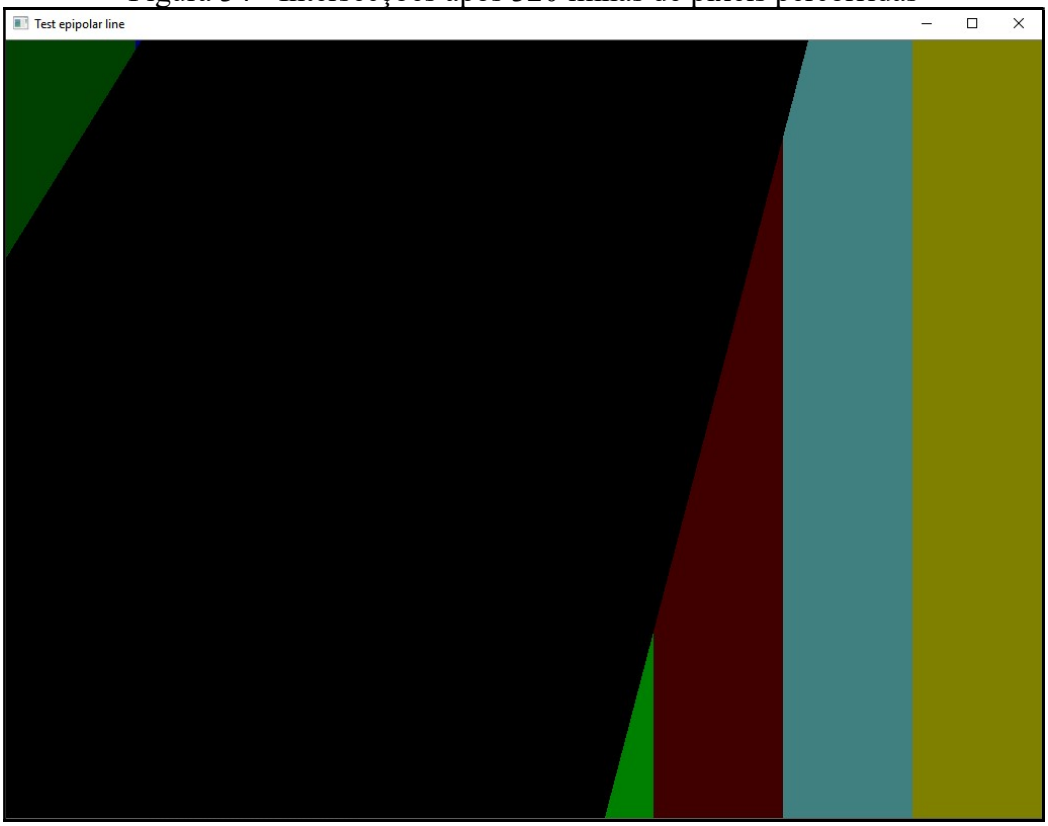
Fonte: elaborado pelo autor.

Figura 33 - Intersecções após 160 linhas de pixels percorridas



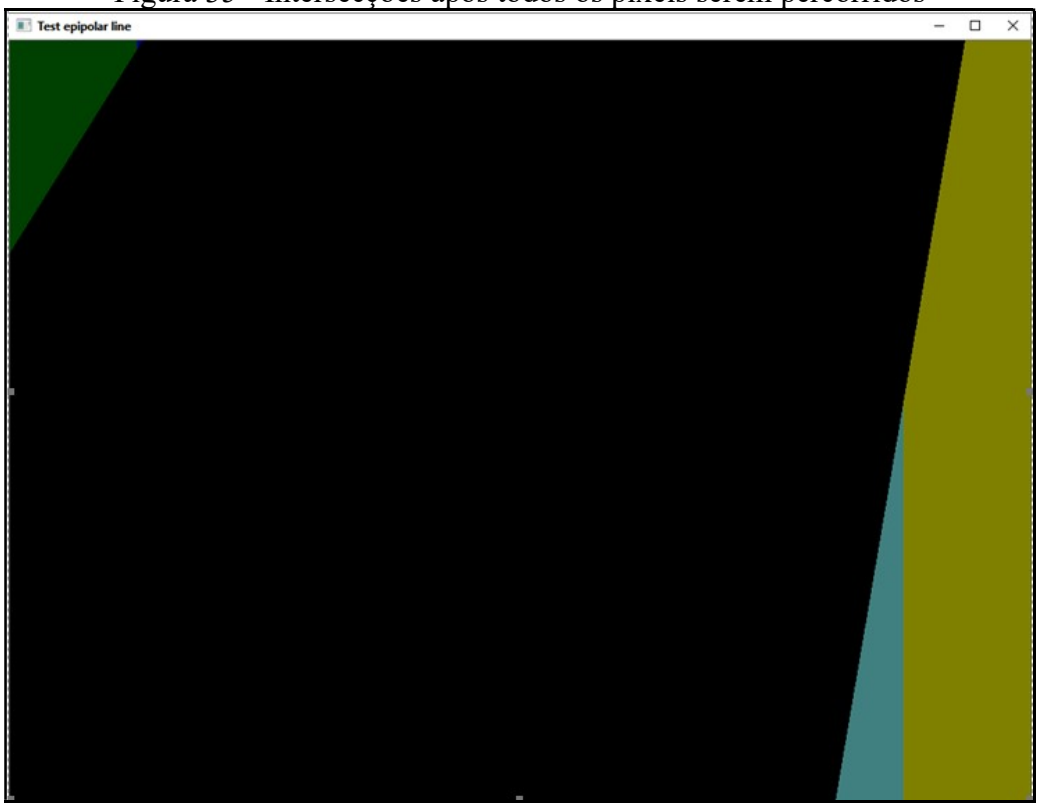
Fonte: elaborado pelo autor.

Figura 34 - Intersecções após 320 linhas de pixels percorridas



Fonte: elaborado pelo autor.

Figura 35 - Intersecções após todos os pixels serem percorridos



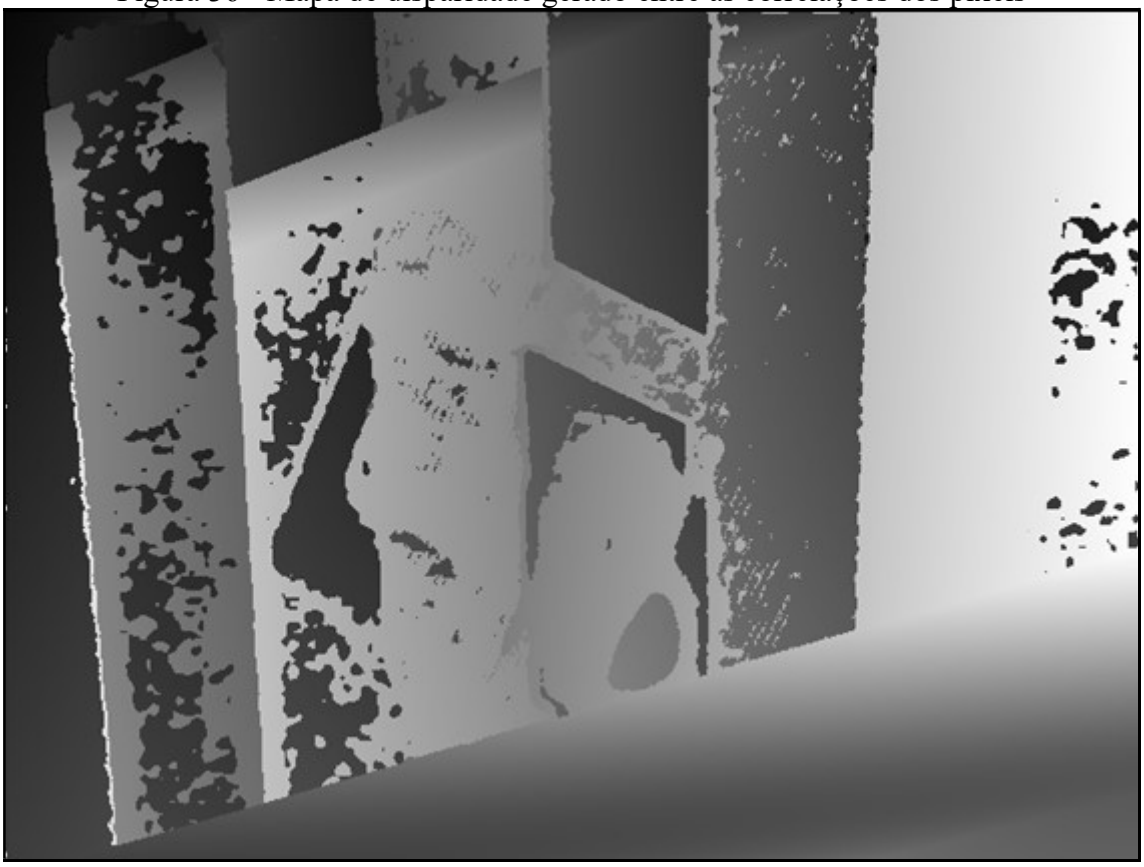
Fonte: elaborado pelo autor.

3.4.3 MAPA DE DISPARIDADE

Para realizar a validação do processo de correlação dos pixels entre as duas imagens, utilizando a fórmula de SSD através da distância euclidiana das cores nos canais `a` e `b`, foi sugerido por Fernandes (2017d) o processo de geração de um mapa de disparidade da imagem que está sendo reconstruída. Na Figura 31 é apresentada a imagem utilizada para realizar a reconstrução.

Conforme imagem Figura 36 do mapa de disparidade, não está sendo possível observar o possível relevo a partir dos pixels correlacionados. Para realizar a observação é necessário levar em consideração apenas as fronteiras entre a luz estruturada emitida na cena. Sendo que nessas fronteiras não está sendo observado nenhuma alteração do relevo, implicando assim em um erro que pode estar ocorrendo ao percorrer os pixels na linha epipolar, ou ao realizar a correlação dos pixels.

Figura 36 - Mapa de disparidade gerado entre as correlações dos pixels



Fonte: elaborador pelo autor.

A geração do mapa de disparidade pode ter seus resultados diminuídos devido ao tamanho de cada coluna da luz estruturada, como a reconstrução só pode ser feita entre as fronteiras o processo é impactado por esse tamanho. Outro ponto que pode estar diminuindo a qualidade da correlação seria o próprio tamanho do *kernel* aplicado ao realizar a fórmula de

SSD. Devido aos mesmos fatores apresentados sobre o tamanho das colunas do padrão estruturado. Sendo ainda o método de correlação fica passível a problemas de iluminação externa e variâncias de cores com base na cor do objeto que é reconstruído. Um algoritmo de programação dinâmica como o proposto por Li, Straub e Prautzsch (2004, p. 2-4) poderia aumentar a qualidade da correlação.

Portanto tem-se então a necessidade de revalidar as etapas correlação dos pixels, como também a etapa de definição da linha epipolar, sendo assim possível identificar o possível erro no mapa de disparidade. Pode-se ainda diminuir o tamanhos das colunas da luz estruturada emitida e o ângulo entre a câmera e o projetor para validar se a correlação é melhorada.

3.4.4 DISPONIBILIZAÇÃO DE UM DATASET DE UM TERRENO PARA O UNITY

Um problema encontrado quanto aos objetivos iniciais do trabalho proposto consiste na etapa de disponibilização de um *dataset* com os dados do modelo 3D, que seria gerado e fornecido em tempo real para a plataforma Unity. O problema encontrado foi que para disponibilizar os dados de um terreno, ou um relevo, dentro do Unity é necessário gerar uma imagem de um mapa de altura. O mapa de altura consiste em uma imagem em escala de cinza onde quanto mais escuro o pixel, mais funda é considerada a profundidade daquele elemento (UNITY TECHNOLOGIES, 2017).

Para ser fornecido o mapa de altura ao Unity é necessário importar este mapa de altura para um objeto do tipo terreno pela interface gráfica da própria plataforma, impossibilitando assim a leitura e atualização em tempo real para a plataforma sendo a necessidade de criar uma aplicação própria para isso (UNITY TECHNOLOGIES, 2017).

Conforme apresentado, este processo impossibilita para que a biblioteca funcione separada do Unity, sendo necessário o desenvolvimento de um script que identifique alterações no arquivo e atualize o terreno em tempo real. Segundo trabalho proposto por Diegoli Neto (2017), prova que é possível manipular e editar o terreno através de scripts do Unity.

4 CONCLUSÕES

Com base nos pontos apresentados no trabalho, os objetivos especificados foram alcançados. Tendo como comparação o algoritmo de calibração de câmera, projetor e ambiente estéreo que manteve a mesma taxa de erro com o mesmo *dataset* de imagens utilizado no software de Moreno e Taubin (2012). Os algoritmos propostos para realizar as intersecções entre as linhas epipolares e as bordas das imagens, foram validados com base nas imagens de testes apresentadas. Tendo sido apresentados resultados que comprovam que a correlação dos pixels está sendo feita através do mapa de disparidade. Possibilitando assim a continuação do desenvolvimento da correção da geometria para a etapa de triangulação.

Quanto ao objetivo de execução em tempo real, ele não pode ser atingido devido aos cálculos executados de forma linear e sem a utilização de processos paralelos na CPU, nem a utilização de GPGPU. Atualmente o algoritmo demora em torno de 1 hora para percorrer e correlacionar todos os pixels, tornando assim necessária a otimização destes cálculos utilizando os conceitos de GPGPU. Tendo como adaptar facilmente ao código atual, já que as tarefas de reconstrução utilizam os conceitos da orientação a objetos para realizar os processos de correlação e reconstrução.

Conforme apresentado os objetivos para carregar os dados do relevo em tempo real para o Unity também não foram alcançados, devido ao não alcance do objetivo de reconstrução em tempo real. Como também a necessidade da criação de scripts específicos para o Unity a fim de ter um terreno atualizado com base em parâmetros externos.

As principais vantagens do trabalho são as facilidades de alterações para otimização do processo de correlação e reconstrução. Outro ponto é o detalhamento dos conteúdos explorados para reconstrução 3D, Geometria Epipolar e GPGPU, sendo todos explorados amplamente na fundamentação teórica. Por fim, outra vantagem do trabalho posso ser visto na remoção da dependência da biblioteca do QT, na adaptação do algoritmo de calibração do ambiente estéreo de Moreno e Taubin (2012), facilitando assim a utilização em outros projetos. A principal desvantagem pode ser vista tanto no tempo necessário para executar todo o processo de correlação, como também em não ter finalizado o processo de geração da nuvem de pontos.

4.1 EXTENSÕES

Sugerem-se as seguintes extensões para trabalhos futuros:

a) ajustar as dependências da biblioteca para que não seja necessário fazer o *link* para elas na aplicação que utilizará a biblioteca;

b) testar diferentes padrões estruturado, com tamanho de colunas menores e cores contrastantes;
c) alterar os nomes dos parâmetros de correlação para não referenciarem aos nomes projetor e câmera, considerar a utilização de usar primeira e segunda imagem;
d) alterar a inicialização da biblioteca para definir os parâmetros no método construtor da classe `ReliefMap`;
e) verificar o processo de carregamento das imagens na pasta de calibração para retornar à quantidade de imagens pertencentes em cada pasta de calibração;
f) desenvolver o processo de captura das imagens seguindo o software apresentado por Moreno e Taubin (2012);
g) reescrever a etapa de correlação das linhas epipolares utilizando GPGPU;
h) reescrever a etapa de correlação dos pixels utilizando o algoritmo apresentado por Li, Straub e Prautzsch (2004, p. 2-4);
i) finalizar processo de correção da geometria para triangulação;
j) gerar nuvem de pontos com base na triangulação a partir da grade regular retangular da imagem capturada pela webcam;
k) gerar um *heightmap* com base na nuvem de pontos obtida;
l) adaptar a biblioteca para ser capaz de informar tipos distintos de modelos 3D finais, como por exemplo OBJ e *heightmap*;
m) atualizar a aplicação de testes da biblioteca para ler e exibir em tempo real o arquivo da nuvem de pontos.

REFERÊNCIAS

BOHN, Noemia. **Caixa e - água.** Blumenau: Furb, 2017. 78 p. Disponível em: <http://www.youblisher.com/p/1715773-e-book/>. Acesso em: 18 dez. 2017.

COLLINS, Robert. **Correspondence Matching.** 2012. Disponível em: <http://www.cse.psu.edu/~rtc12/CSE486/lecture07.pdf>. Acesso em: 28 de Novembro de 2017.

DIEGOLI NETO, Guilherme. **Simulação de dinâmica do relevo através da transformação do mapa de altura**. 2017. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

FERNANDES, Leandro Augusto Frata. **TCC 2017 II** [mensagem pessoal]. Mensagem recebida por <kevinstortz.ks@gmail.com> em 28 de Abril de 2017?a.

FERNANDES, Leandro Augusto Frata. **Correlação das linhas epipolares** [mensagem pessoal]. Mensagem recebida por <kevinstortz.ks@gmail.com> em 10 de Novembro de 2017?b.

FERNANDES, Leandro Augusto Frata. **Correlação dos pixels e aplicação do raio entre eles** [mensagem pessoal]. Mensagem recebida por <kevinstortz.ks@gmail.com> em 24 de Novembro de 2017?c.

FERNANDES, Leandro Augusto Frata. **Validação da correlação dos pixels** [mensagem pessoal]. Mensagem recebida por <kevinstortz.ks@gmail.com> em 28 de Novembro de 2017?d.

FERNANDES, Leandro Augusto Frata. **Estudo de Métodos para Extração de Formas e Realização de Medidas a Partir de Imagens.** 2005. 32 p. Trabalho Individual I (Programa de Pós-Graduação em Informática) - Universidade Federal do Rio Grande do Sul. Disponível em: <http://www2.ic.uff.br/~laffernandes/projects/metrology/2005_UFRGS_TI/fernandes_TI_1159.pdf>. Acesso em: 22 de Maio de 2017.

HARTLEY, Richard; ZISSERMAN, Andrew. **Multiple View Geometry in Computer Vision.** 2. ed. Cambridge: Cambridge University Press, 2004. 655 p.

LI, Hao; STRAUB, Raphael; PRAUTZSCH, Hartmut. Fast subpixel accurate reconstruction using structured light. **Visualization, Imaging, And Image Processing,** Marbella, v. 452, n. 275, p.396-401, 06 set. 2004. Anual. Disponível em: <http://www.hao-li.com/publications/papers/viip2004FSA.pdf>. Acesso em: 10 de Maio de 2017.

MANCINI, Francesco, et al. Using Unmanned Aerial Vehicles (UAV) for high-resolution reconstruction of topography: the Structure from Motion approach on coastal environments. **Remote Sens,** Basel, v.5, n.12, p. 6880-6898 2013. Disponível em: <http://www.mdpi.com/2072-4292/5/12/6880>. Acesso em: 25 de Março de 2017.

MENDONÇA, Cláudio. **Topografia (1):** Hipsometria e Curvas de Nível. São Paulo. 2007. Disponível em: <https://educacao.uol.com.br/disciplinas/geografia/topografia-1-hipsometria-e-curvas-de-nivel.htm>. Acesso em: 30 de Março de 2017.

MORENO, Daniel; TAUBIN, Gabriel. **Simple, Accurate, and Robust Projector-Camera Calibration.** Providence: Brown University, 2012. 8 p. Disponível em: <http://mesh.brown.edu/calibration/files/Simple, Accurate, and Robust Projector-Camera Calibration.pdf>. Acesso em: 10 de Outubro 2017.

MUNSELL, Albert. **Munsell Manual of Color:** Defining and Explaining the Fundamental Characteristics of Color. Baltimore: Waverly Press Inc., 1929. 33 p. Disponível em: <http://munsell.com/wp-content/uploads/2017/03/munsell-manual-of-color.pdf>. Acesso em: 25 de Novembro de 2017.

MYCROFT, Alan. **Programming Language Design and Analysis motivated by Hardware Evolution.** Cambridge: Cambridge University Reporter, 2005. 6 p. Disponível em: <https://www.cl.cam.ac.uk/~am21/papers/sas07final.pdf>. Acesso em: 02 de Outubro 2017.

OPENCV. **Camera Calibration and 3D Reconstruction.** 2017. Disponível em: <https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html>. Acesso em: 26 de Outubro 2017.

OPENCV. **Depth Map from Stereo Images.** 2014. Disponível em: <https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_calib3d/py_depthmap/py_depthmap.html>. Acesso em: 28 de Novembro de 2017.

PENA, Rodolfo F. Alves. **O que é Relevo?.** Brasil Escola. [S.l.], 2017. Disponível em: <http://brasilescola.uol.com.br/o-que-e/geografia/o-que-e-relevo.htm>. Acesso em: 23 de Maio de 2017.

PEREIRA, Adalberto Alves; THOMAZ, Edivaldo Lopes. Hipsometria e Declividade da Bacia Hidrográfica do Arroio Palmeirinha, município de Reserva – PR, utilizando o Software Spring. In: SIMPÓSIO BRASILEIRO DE SENSORIAMENTO REMOTO, 16., 2013, Foz do Iguaçu. **Anais...** . Foz do Iguaçu: Sbsr, 2013. v. 32, p. 3494 - 3501. Disponível em: <http://www.dsr.inpe.br/sbsr2013/files/p0071.pdf>. Acesso em: 27 de Maio de 2017.

PUTTEMANS, Steven (Org.). **Chessboard.** 2015. Disponível em: <https://github.com/opencv/opencv/blob/master/samples/data/chessboard.png>. Acesso em: 30 nov. 2017.

RONCA, Debra. **How to Read a Topographic Map.** 2009. Disponível em: <http://adventure.howstuffworks.com/outdoor-activities/hiking/how-to-read-a-topographic-map2.htm>. Acesso em: 28 de Maio de 2017.

THE MATHWORKS. **Stereo Calibration App.** 2017. Disponível em: <https://www.mathworks.com/help/vision/ug/stereo-camera-calibrator-app.html#buejbxy-3>. Acesso em: 28 de Novembro de 2017.

UNITY TECHNOLOGIES. Unity manual.[S.I.], 2017. Disponível em:< http://docs.unity3d.com/Manual/index.html>. Acesso em: 20 de Setembro de 2016.

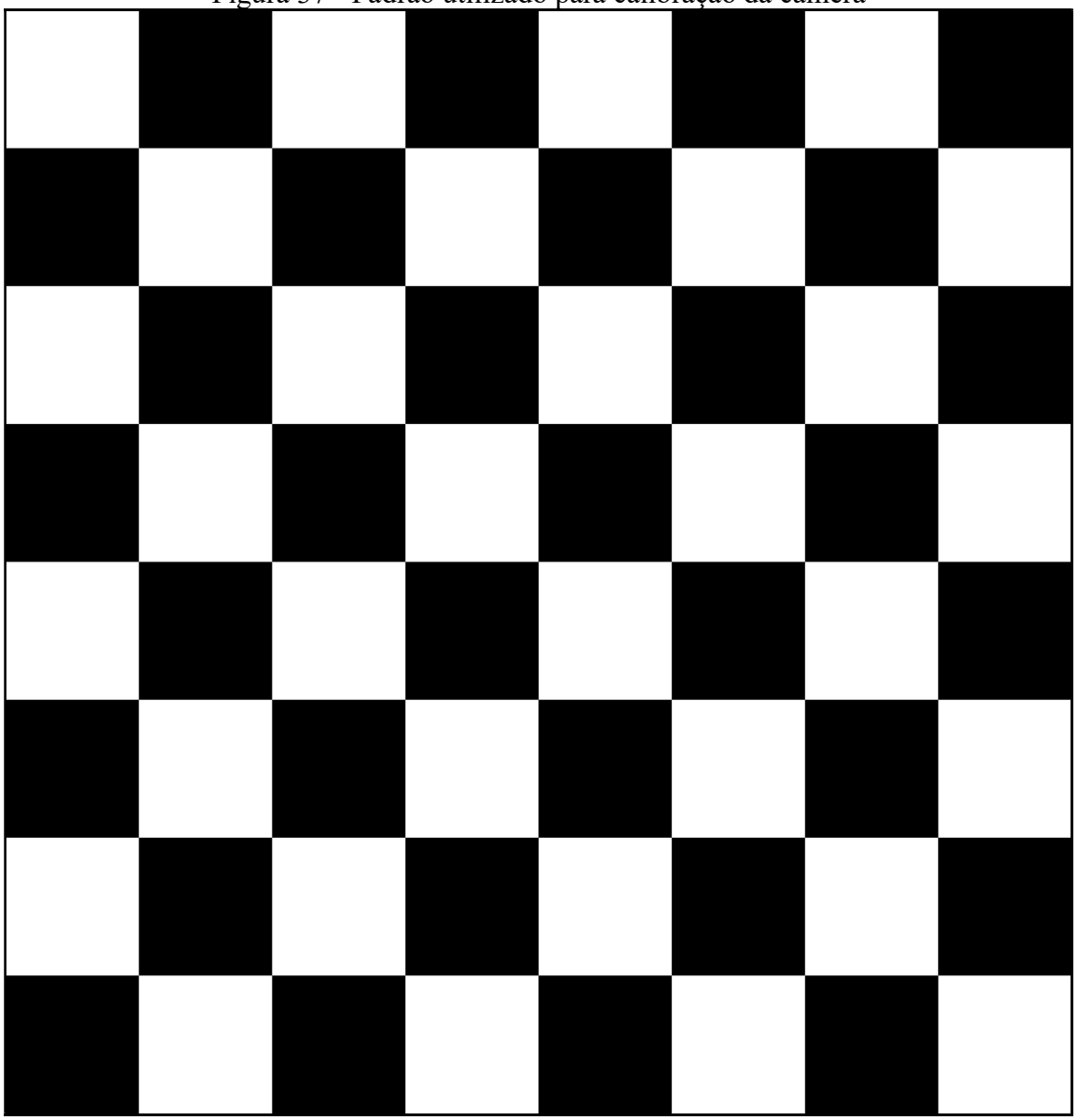
ZAAIJER, Simon. **GPU Based Generation and Real-Time Rendering of Semi-Procedural Terrain Using Features.** 2013. 47 p. Dissertação (Mestrado em Informática) – Leiden Institute of Advanced Computer Science (LIACS), Leiden. Disponível em: <http://liacs.leidenuniv.nl/assets/Masterscripties/2013-11SimonZaaijer.pdf>. Acesso em: 14 de Março de 2017.

ZHANG, Zhengyou. A Flexible New Technique for Camera Calibration. **Ieee Transactions On Pattern Analysis And Machine Intelligence.** [S.I], p. 1330-1334. nov. 2000. Disponível em: < https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr98-71.pdf>. Acesso em: 25 de Outubro de 2017.

ANEXO A – IMAGEM DO TABULEIRO DE XADREZ PARA CALIBRAÇÃO DA CÂMERA FORNECIDA PELO BIBLIOTECA OPENCV

Neste anexo é disponibilizado o arquivo utilizado para realizar a calibração das câmeras, conforme Figura 37, o arquivo consiste da representação de um tabuleiro de xadrez. A biblioteca OpenCV fornece este arquivo entre os arquivos do código fonte.

Figura 37 - Padrão utilizado para calibração da câmera



Fonte: Puttemans (2015).