Biblioteca para reconstrução de relevos

Aluno(a): Kevin Stortz

Orientador: Dalton Solano dos Reis

FURB
UNIVERSIDADE DE BLUMENAU

Roteiro

- Introdução
- Objetivos
- Fundamentação
- Correlatos
- Requisitos/especificação
- Implementação
- Resultados/conclusões

Introdução

- Topologias e relevos
- GPGPU
- Calibração de câmeras e calibração estéreo
- Geometria Epipolar
- Correlação de pixels

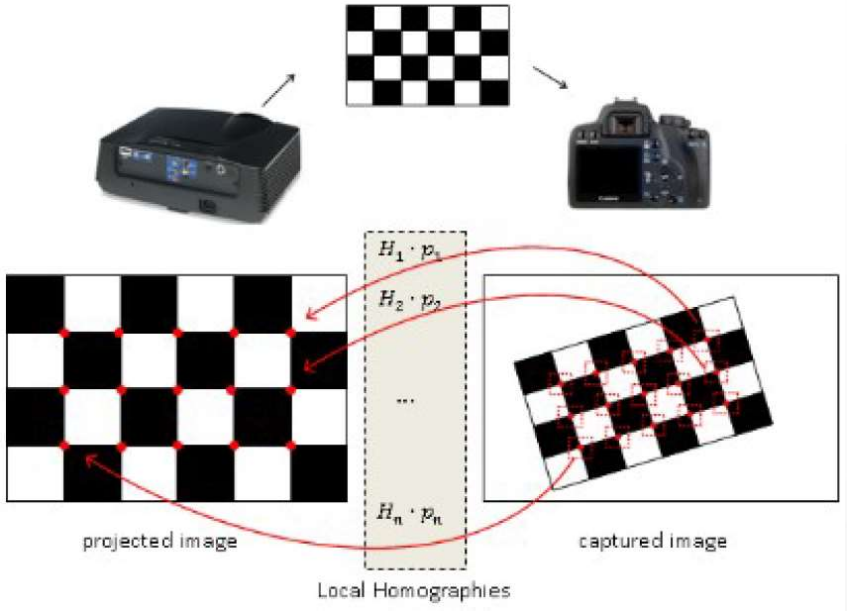
Objetivos

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

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

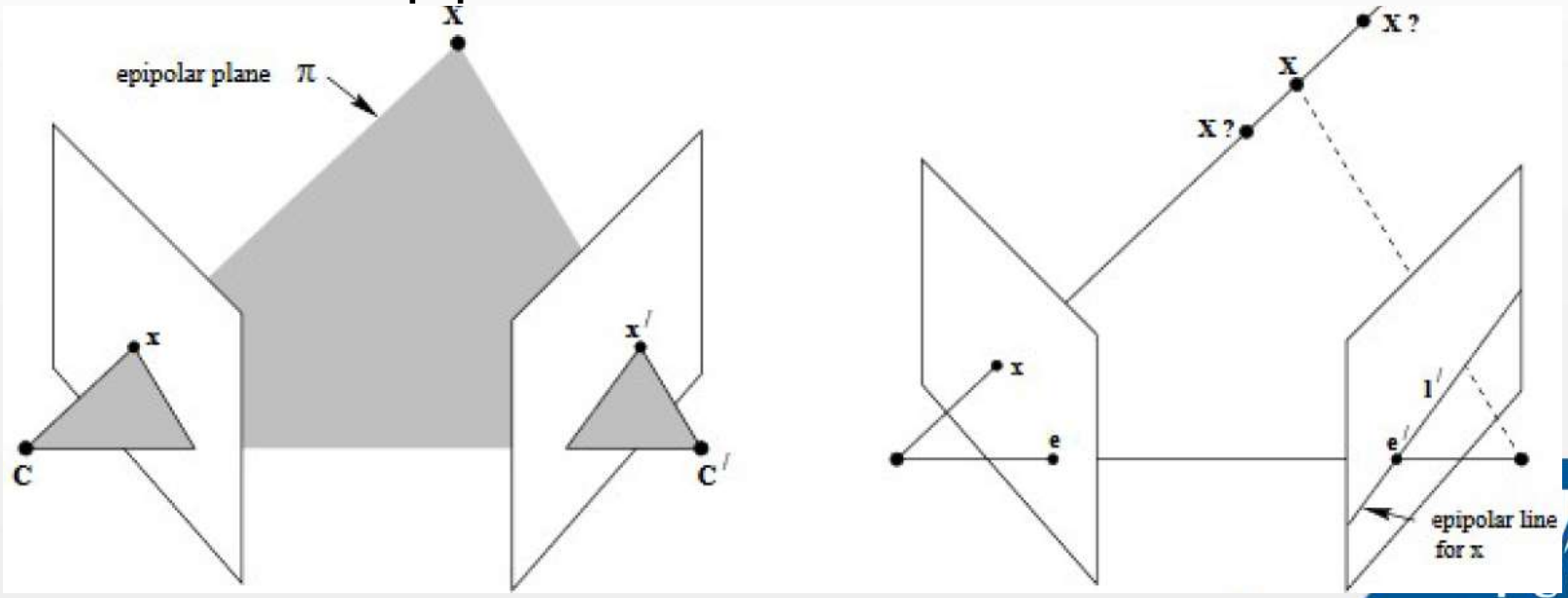
Fundamentação Teórica

- Cartas topográficas
  - Carta Hipsométrica
  - Curvas de Nível
- GPGPU (CUDA)
- Reconstrução 3D
  - Calibração de Câmeras
    - WebCam
    - Projetor
    - Estéreo



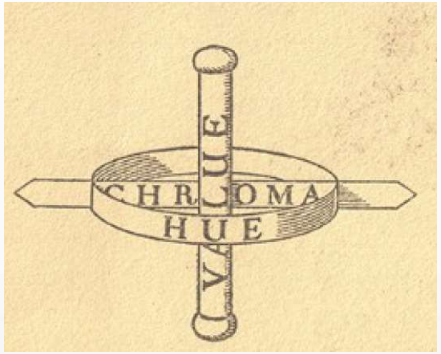
Fundamentação Teórica

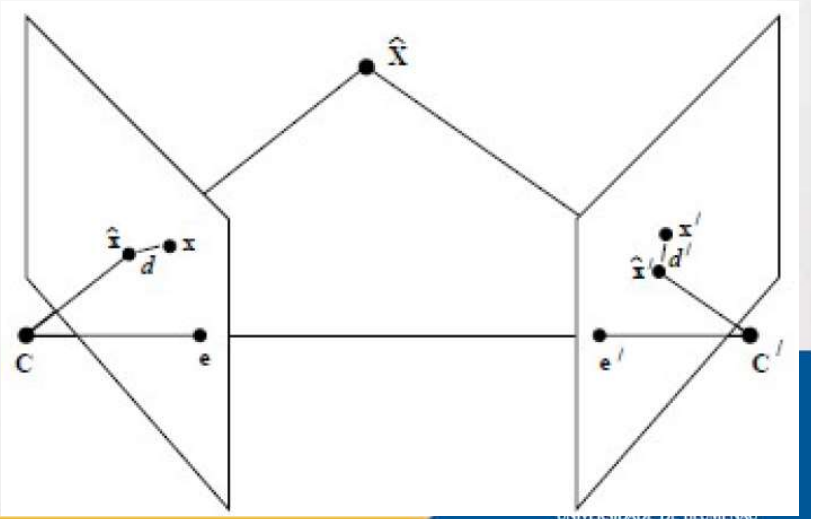
– Geometria Epipolar
  - Matriz Fundamental
  - Epipolos
  - Linha Base
  - Linha Epipolar



Fundamentação Teórica

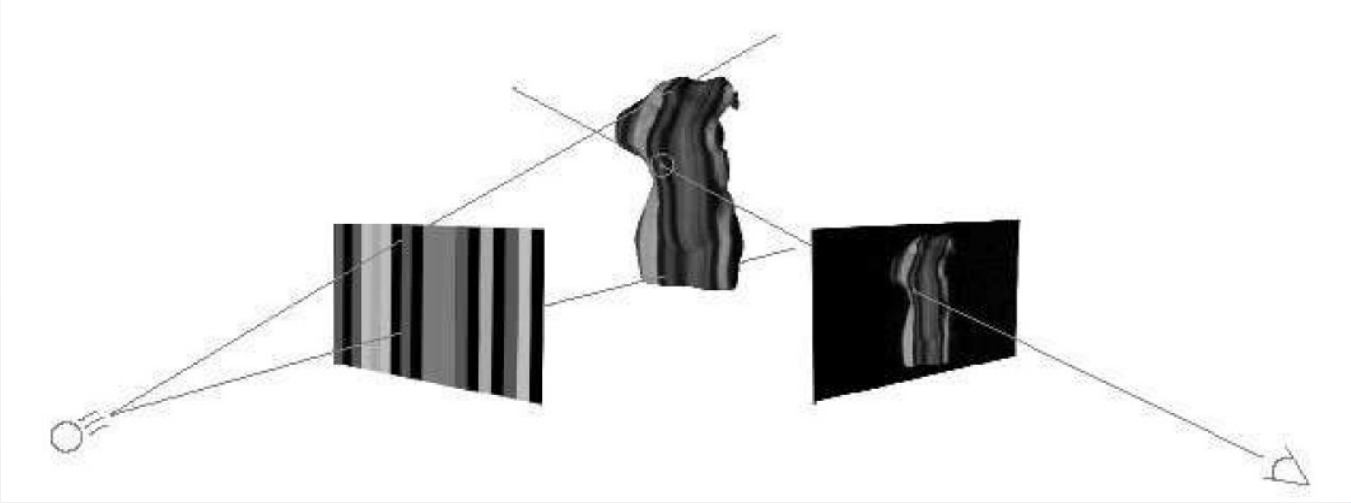
– Triangulação
  - Identificar as intersecções da linha epipolar com as bordas da imagem
    – Sequencia de produto vetorial
  - Correlacionar os pixels
    – Teoria das cores de Munsell
    – SSD
    – Mapa de disparidade
    – Corrigir o erro da geometria





Trabalhos Correlatos

- Reconstrução 3D
- Calibração de câmera
- Ambiente estéreo
- Correlação de pixels



Li, Straub e Prautzsch, 2004

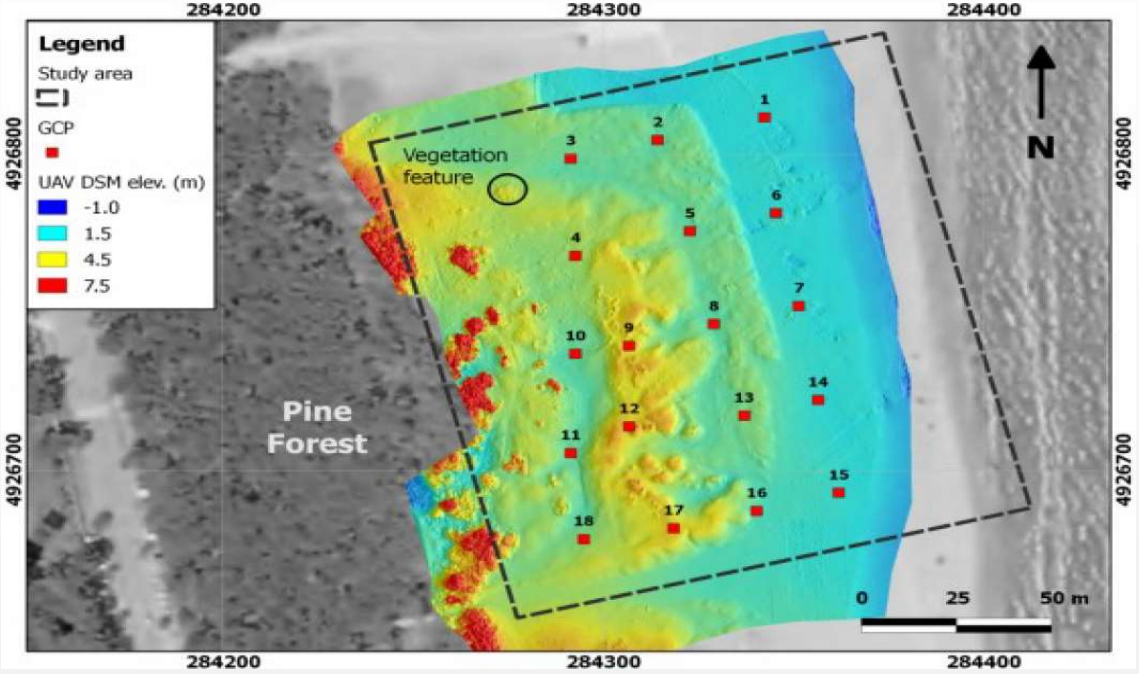
Trabalhos Correlatos

- Reconstrução 3D
- GPGPU
- Geração de malha do terreno

Zaaijer, 2013

Trabalhos Correlatos

- Reconstrução 3D – VANTs
- DSM Múltiplas imagens
- Photoscan



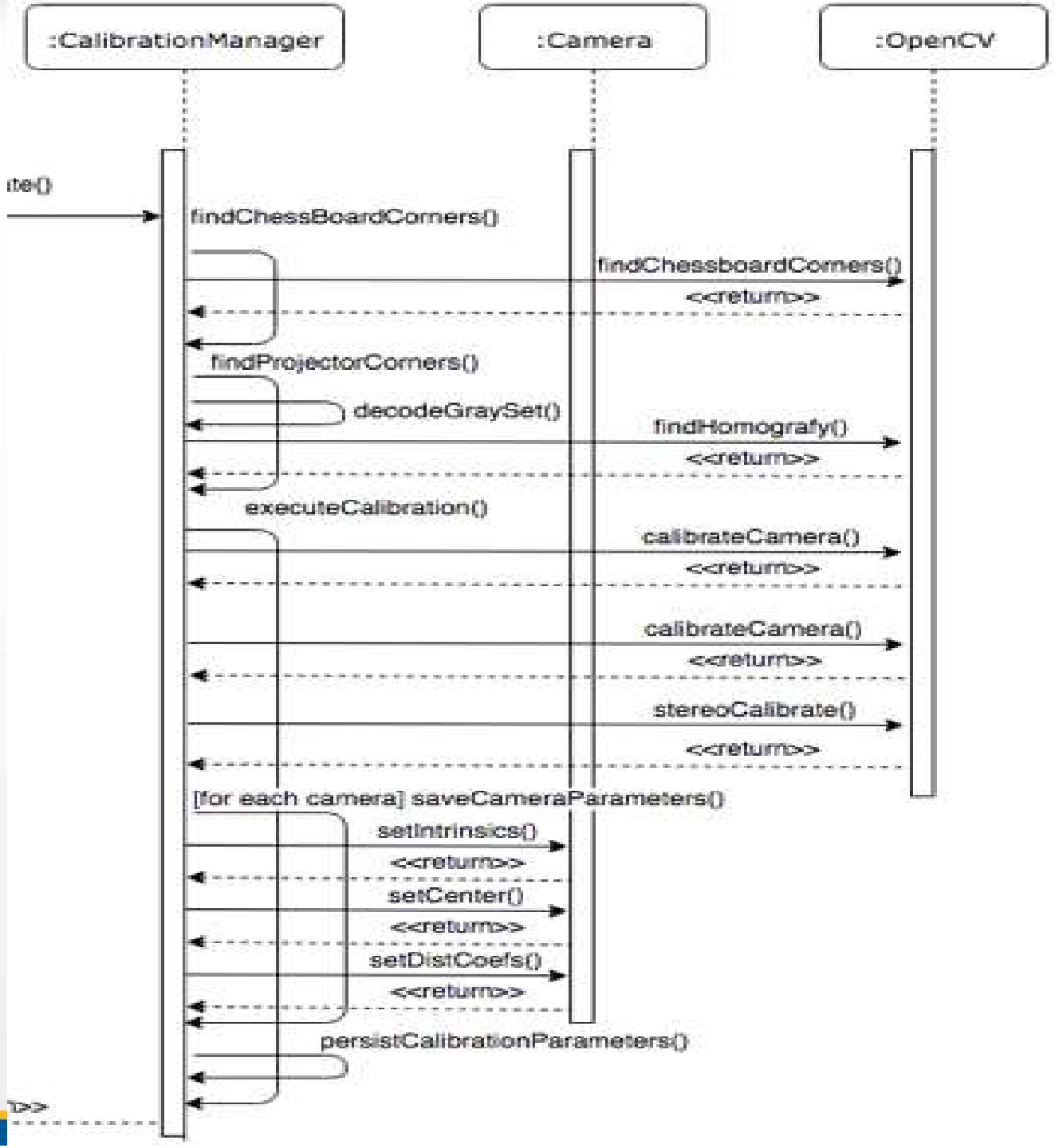
Mancini et al, 2013

FURB
UNIVERSIDADE DE BLUMENAU

Requisitos

- RF01 - realizar a calibração da câmera e do projetor para a remoção da distorção
- RF04 - realizar a intersecção das linhas epipolares com as bordas da segunda imagem
- RF05 - realizar a correlação do pixel da câmera com o correlato na imagem do projetor
- RF06 - gerar o mapa de disparidade para validar a correlação dos pixels
- RNF01 - ser desenvolvido em C++
- RNF04 - ser desenvolvida com suporte a multiplataforma de arquitetura e sistema operacional
- RNF05 - utilizar o ambiente de desenvolvimento Visual Studio 2015
- RNF07 - ser desenvolvida utilizando a biblioteca OpenCV

Sequencias: Calibração



Sequencias: Correlação de pixels

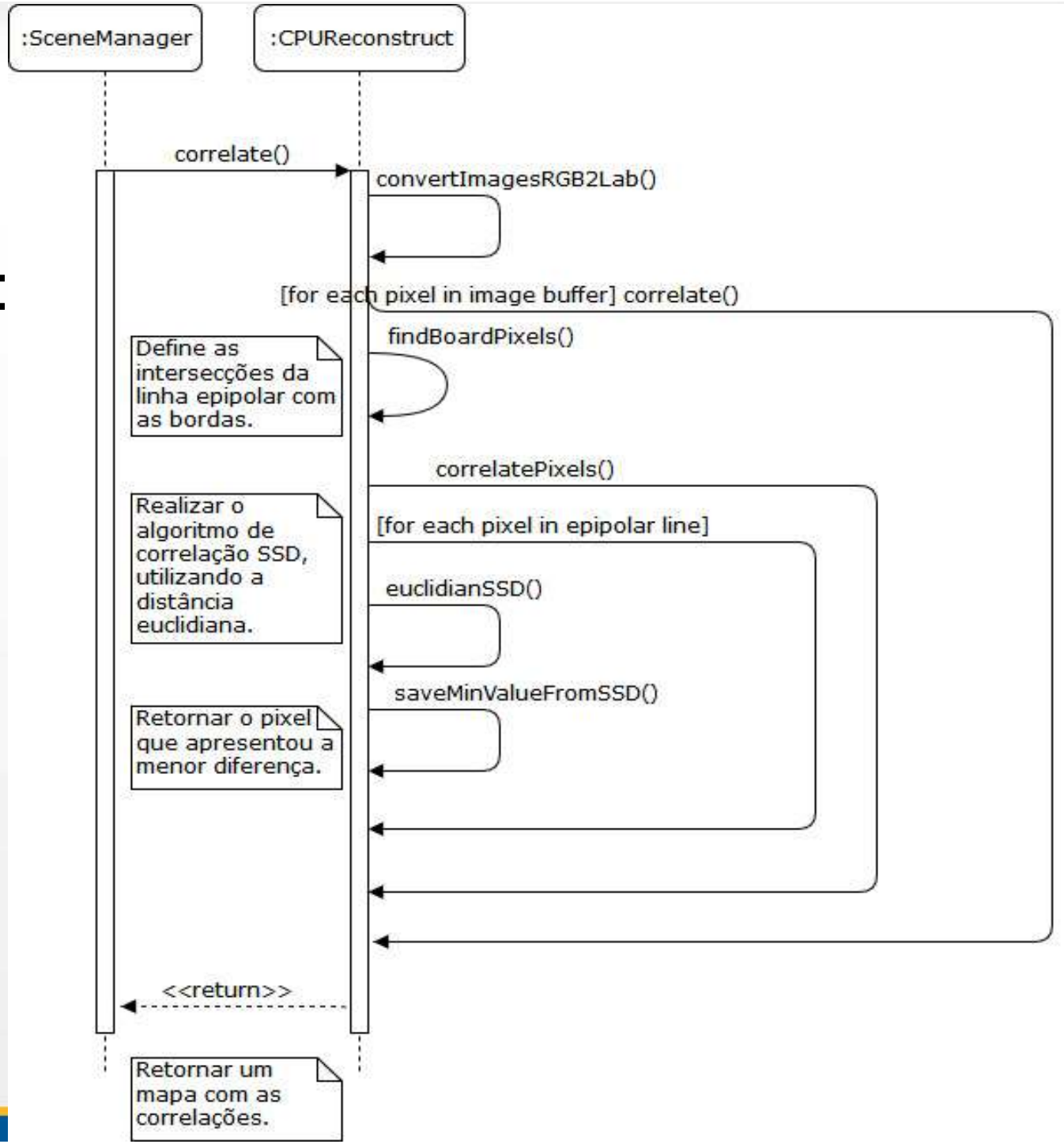
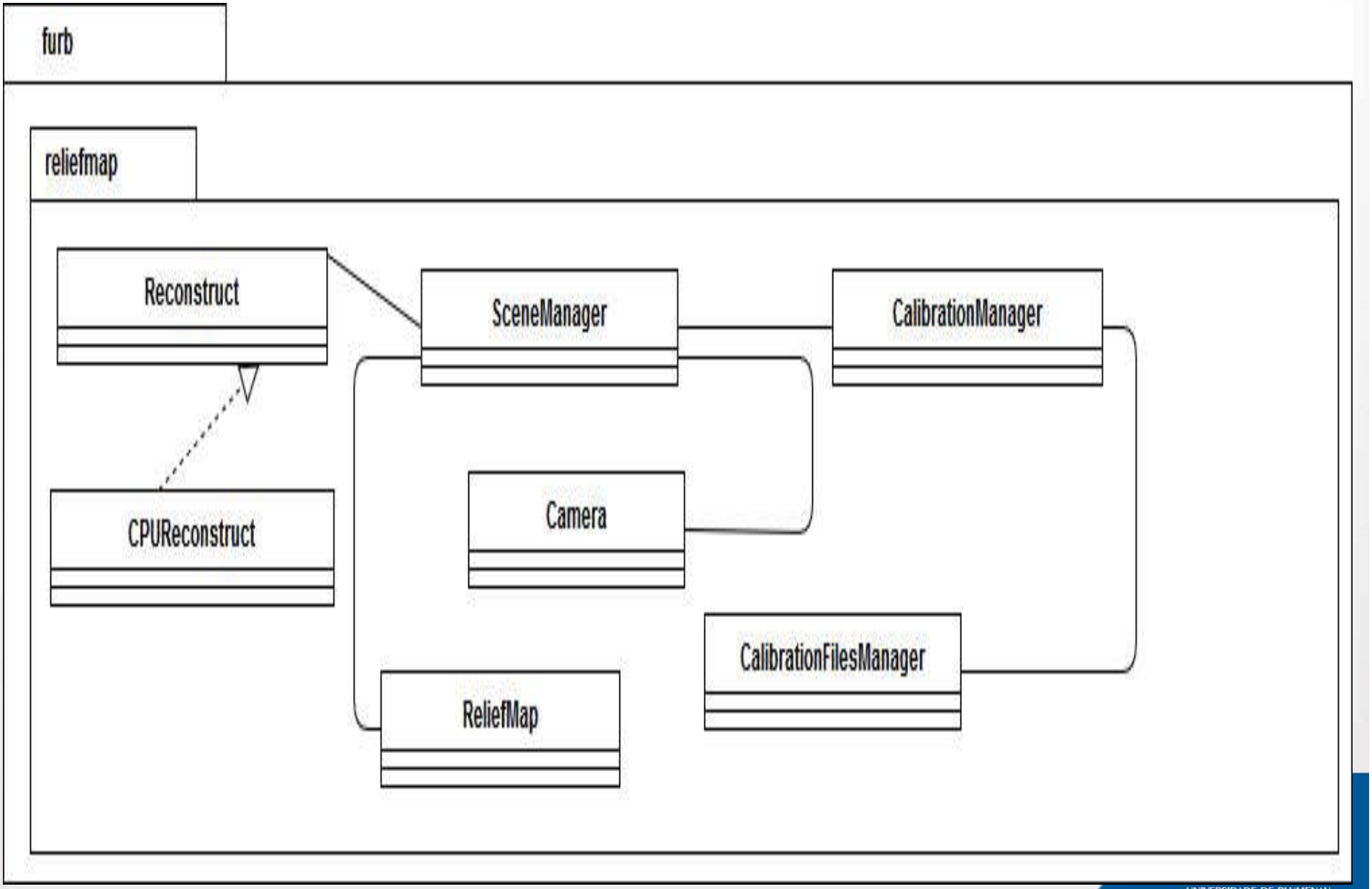


Diagrama de classes



Implementação

- Calibração de câmera
- Calibração do projetor e estéreo
- Intersecções com a linha epipolar
- Correlação dos pixels
- Mapa de disparidade

Calibração da câmera

```
130    std::vector<cv::Point2f> & camCorners = this->_cornersCamera->at(i);
131    std::vector<cv::Point3f> & worldCorners = this->_cornersWorld->at(i);
132
133    if (cv::findChessboardCorners(small_img, cornerCount, camCorners,
134        cv::CALIB_CB_ADAPTIVE_THRESH + cv::CALIB_CB_NORMALIZE_IMAGE)) {
135
136        std::cout << " - corners: " << camCorners.size() << std::endl;
137        //generate world object coordinates
138        for (int h = 0; h<cornerCount.height; h++) {
139            for (int w = 0; w<cornerCount.width; w++) {
140                worldCorners.push_back(cv::Point3f(cornerSize.width * w, cornerSize.height * h, 0.f));
141            }
142        }
143
144    }
```

Calibração Projetor

```
326     std::list<unsigned> directComponentImages;
327     for (unsigned i = 0; i<directLightCount; i++) {
328         int index = totalImages - totalPatterns - directLightCount - directLightOffset + i + 1;
329         directComponentImages.push_back(index);
330         directComponentImages.push_back(index + totalPatterns);
331     }
332
333     for each (unsigned i in directComponentImages){
334         images.push_back(getImage(level, i));
335     }
336     cv::Mat directLight = sl::estimate_direct_light(images, b);
337
338     std::vector<std::string> imageNames;
339     unsigned levelCount = static_cast<unsigned>(this->_calibrationFilesManager.getImageCount(level));
340     for (unsigned i = 1; i<=levelCount; i++) {
341         std::string filename = this->_calibrationFilesManager.getImagePath(level, i);
342         std::cout << "[decode_set " << level << "] Filename: " << filename << std::endl;
343
344         imageNames.push_back(filename);
345     }
346
347     bool rv = sl::decode_pattern(imageNames, patternImage, minMaxImage, projectorSize,
348         sl::RobustDecode | sl::GrayPatternDecode, directLight, m);
349     return rv;
```

FURB
UNIVERSIDADE DE BLUMENAU

```
234  unsigned WINDOW_SIZE = 60 / 2;
235  std::vector<cv::Point2f> imgPoints, projPoints;
236  if (p.x>WINDOW_SIZE && p.y>WINDOW_SIZE && p.x + WINDOW_SIZE<patternImage.cols && p.y + WINDOW_SIZE<patternImage.rows) {
237      for (unsigned h = p.y - WINDOW_SIZE; h<p.y + WINDOW_SIZE; h++) {
238          register const cv::Vec2f * row = patternImage.ptr<cv::Vec2f>(h);
239          register const cv::Vec2b * minMaxRow = minMaxImage.ptr<cv::Vec2b>(h);
240          //cv::Vec2f * out_row = out_pattern_image.ptr<cv::Vec2f>(h);
241          for (unsigned w = p.x - WINDOW_SIZE; w<p.x + WINDOW_SIZE; w++) {
242              const cv::Vec2f & ptn = row[w];
243              const cv::Vec2b & minMax = minMaxRow[w];
244              //cv::Vec2f & out_pattern = out_row[w];
245              if (sl::INVALID(ptn)) {
246                  continue;
247              }
248              if ((minMax[1] - minMax[0])<static_cast<int>(threshold)) {   //apply threshold and skip
249                  continue;
250              }
251
252              imgPoints.push_back(cv::Point2f(w, h));
253              projPoints.push_back(cv::Point2f(ptn));
254
255              //out_pattern = pattern;
256          }
257      }
258      cv::Mat H = cv::findHomography(imgPoints, projPoints, CV_RANSAC);
259      //std::cout << " H:\n" << H << std::endl;
260      cv::Point3d Q = cv::Point3d(cv::Mat(H*cv::Mat(cv::Point3d(p.x, p.y, 1.0))));
261      q = cv::Point2f(Q.x / Q.z, Q.y / Q.z);
262  }
```

Calibração Estéreo

- Normalização das posições do objetos locais e referentes ao mundo

```
452  cv::Size camSize = cv::Size(640, 480);
453  std::vector<cv::Mat> camRvecs, camTvecs;
454  double camError = cv::calibrateCamera(worldCornersActive, cameraCornersActive, camSize, camIntrinsics,
455      camDistCoeffs, camRvecs, camTvecs, calFlags, cv::TermCriteria(cv::TermCriteria::COUNT + cv::TermCriteria::EPS, 50, DBL_EPSILON));
456  std::cout << "Camera error: " << camError << std::endl;
457
458  //calibrate the projector
459  std::vector<cv::Mat> projRvecs, projTvecs;
460  double projError = cv::calibrateCamera(worldCornersActive, projectorCornersActive, projectorSize, projIntrinsics,
461      projDistCoeffs, projRvecs, projTvecs, calFlags, cv::TermCriteria(cv::TermCriteria::COUNT + cv::TermCriteria::EPS, 50, DBL_EPSILON));
462  std::cout << "Projector error: " << projError << std::endl;
463
464  cv::Mat R, T, E, F;
465  double stereoError = cv::stereoCalibrate(worldCornersActive, cameraCornersActive, projectorCornersActive, camIntrinsics, camDistCoeffs, projIntrinsics, projDistCoeffs,
466      camSize, R, T, E, F, cv::CALIB_FIX_INTRINSIC + calFlags, cv::TermCriteria(cv::TermCriteria::COUNT + cv::TermCriteria::EPS, 150, DBL_EPSILON));
467  std::cout << "Stereo error: " << stereoError << std::endl;
468
469  return this->saveCameraParameters(camera1, camIntrinsics, camDistCoeffs) && this->saveCameraParameters(camera2, projIntrinsics, projDistCoeffs) &&
470      this->persistCalibrationParameters(camIntrinsics, camDistCoeffs, projIntrinsics, projDistCoeffs, R, T, camError, projError, stereoError, F);
```

Intersecção da linha epipolar

```
61  inline void defineProjectorBounds(const cv::Mat& projectorPattern) {
62      mathfu::Vector<double, 3> topLeft(0, 0, 1);
63      mathfu::Vector<double, 3> topRight(projectorPattern.cols - 1, 0, 1);
64      mathfu::Vector<double, 3> bottomLeft(0, projectorPattern.rows - 1, 1);
65      mathfu::Vector<double, 3> bottomRight(projectorPattern.cols - 1, projectorPattern.rows - 1, 1);
66
67      this->_top = mathfu::Vector<double, 3>::CrossProduct(topLeft, topRight);
68      this->_right = mathfu::Vector<double, 3>::CrossProduct(topRight, bottomRight);
69      this->_bottom = mathfu::Vector<double, 3>::CrossProduct(bottomRight, bottomLeft);
70      this->_left = mathfu::Vector<double, 3>::CrossProduct(bottomLeft, topLeft);
71  }
```

```
135 void CPUReconstruct::findBorderPixels(const mathfu::Vector<double, 3>& epipolarLine) {
136     // acha as intersecções das bordas com a linha epipolar
137     mathfu::Vector<double, 3> intersectTop = mathfu::Vector<double, 3>::CrossProduct(epipolarLine, this->_top);
138     this->normalizeVector(intersectTop);
139     mathfu::Vector<double, 3> intersectRight = mathfu::Vector<double, 3>::CrossProduct(epipolarLine, this->_right);
140     this->normalizeVector(intersectRight);
141     mathfu::Vector<double, 3> intersectBottom = mathfu::Vector<double, 3>::CrossProduct(epipolarLine, this->_bottom);
142     this->normalizeVector(intersectBottom);
143     mathfu::Vector<double, 3> intersectLeft = mathfu::Vector<double, 3>::CrossProduct(epipolarLine, this->_left);
144     this->normalizeVector(intersectLeft);
145
146     // encontra agora analisando as linhas de intersecção inferior e superior com as das laterais..
147     if (intersectBottom.x > intersectLeft.x && intersectBottom.x < intersectRight.x) {
148         this->defineIntersection(intersectBottom, this->_firstPixelIntersect);
149     } else if (intersectBottom.x < intersectLeft.x) {
150         this->defineIntersection(intersectLeft, this->_firstPixelIntersect);
151     } else if (intersectBottom.x < intersectRight.x) {
152         this->defineIntersection(intersectRight, this->_firstPixelIntersect);
153     }
154
155     if (intersectTop.x > intersectLeft.x && intersectTop.x < intersectRight.x) {
156         this->defineIntersection(intersectTop, this->_secondPixelIntersect);
157     }
158     else if (intersectTop.x < intersectLeft.x) {
159         this->defineIntersection(intersectLeft, this->_secondPixelIntersect);
160     }
161     else if (intersectTop.x < intersectRight.x) {
162         this->defineIntersection(intersectRight, this->_secondPixelIntersect);
163     }
164 }
```

Correlação de pixels

```
103     int middleHeight = HEIGHT_MASK / 2;
104     int middleWidth = WIDTH_MASK / 2;
105     for (int i = 0; i < HEIGHT_MASK; ++i) {
106         int footsHeigth = i - middleHeight;
107         int newXF = centralPixelF.x + footsHeigth;
108         int newXG = centralPixelG.x + footsHeigth;
109 
110         if (newXF >= 0 && newXF < labImageF.rows &&
111             newXG >= 0 && newXG < labImageG.rows) {
112             for (int j = 0; j < WIDTH_MASK; ++j) {
113                 int footsWidth = j - middleWidth;
114                 int newYF = centralPixelF.y + footsWidth;
115                 int newYG = centralPixelG.y + footsWidth;
116 
117                 if (newYF >= 0 && newYF < labImageF.cols &&
118                     newYG >= 0 && newYG < labImageG.cols) {
119                     cv::Vec3b labPixelF = labImageF.at<cv::Vec3b>(newXF, newYF);
120                     cv::Vec3b labPixelG = labImageG.at<cv::Vec3b>(newXG, newYG);
121 
122                     double a = labPixelF.val[1] - labPixelG.val[1];
123                     double b = labPixelF.val[2] - labPixelG.val[2];
124                     double euclidianDist = a * a + b * b;
125 
126                     sum += euclidianDist *euclidianDist;
127                 }
128             }
129         }
130     }
```

Geração do mapa de disparidade

```
190  cv::Mat img(imageSize, CV_8UC1, cv::Scalar(255));
191  for (int i = 0, size = distances.size(); i < size; ++i) {
192      std::tuple<cv::Point, int> distancy = distances[i];
193      cv::Point pixel = std::get<0>(distancy);
194      if (pixel.x < imageSize.height && pixel.y < imageSize.width) {
195          double normalized = (std::get<1>(distancy) - min) / (max - min);
196          uchar pixelValue = normalized * 255;
197          img.at<uchar>(pixel.x, pixel.y) = pixelValue;
198      }
199  }
200
201  cv::imwrite("disparityImage.jpg", img);
```

Operacionalidade da Implementação

```
8   #include "ReliefMap.h"
9   #include "SampleGPU.h"
10
11  int main(int argc, char **argv) {
12      furb::reliefmap::ReliefMap library;
13      library.calibrate(1);
14      library.init(1);
15      library.stop();
16
17      return 0;
18  }
```

FURB
UNIVERSIDADE DE BLUMENAU

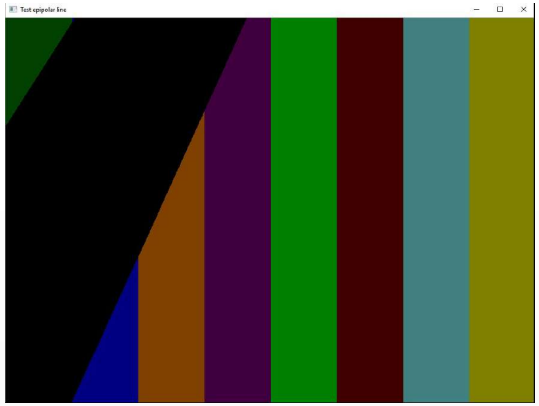
Resultados e Discussões

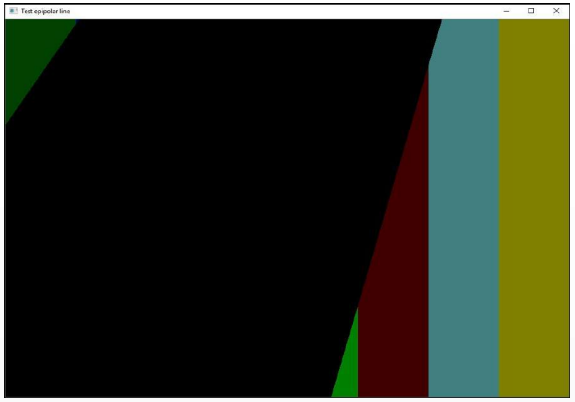
Taxa de erro	Moreno e Taubin (2012)	Algoritmo adaptado pelo autor
Calibração camera	0.403407	0.402455
Calibração projetor	0.766335	0.704214
Calibração estéreo	0.677827	0. 633276

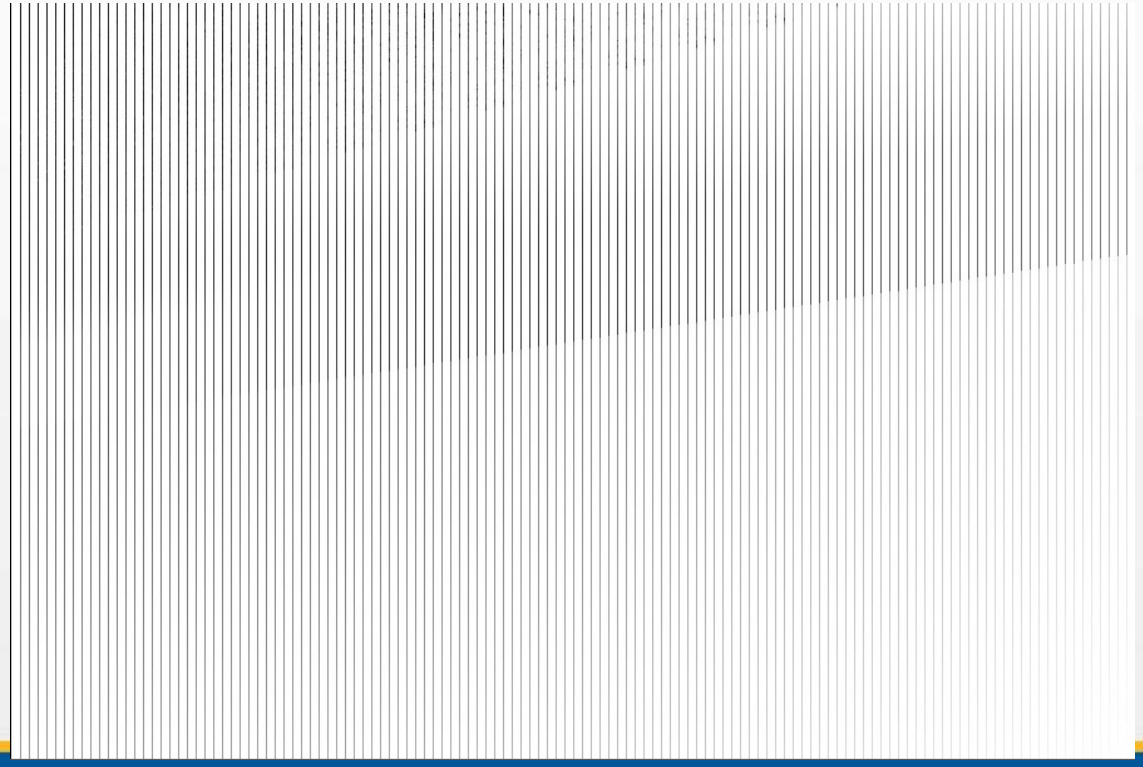
Características	Li, Straub e Prautzsch (2004)	Zaaijer (2013)	Mancini et al (2013)	Trabalho proposto
calibração da câmera e calibração estéreo	Sim	Não	Não	Sim
reconhecimento de relevos	Sim	Sim	Sim	Sim
reconhecimento de relevos por imagens	Sim	Não	Sim	Sim
otimização na GPU	Não	Sim	Não	Não
construção de modelo 3D	Sim	Sim	Não	Não
geração de nuvem de pontos	Sim	Não	Sim	Não

FURB
UNIVERSIDADE DE BLUMENAU

Resultados e Discussões







Conclusões

- Calibração do ambiente estéreo
- Intersecção das linhas epipolares
- Correlação dos pixels
- Geração do mapa de disparidade

Sugestões

- Desenvolver processo de captura das imagens
- Realizar novos testes na geração do mapa de disparidade para comprovar a correlação dos pixels
- Correção da geometria
- Testes com configuração distinta do ambiente estéreo
- Utilização de GPGPU para otimização
- Geração da nuvem de pontos e novos formatos de objetos 3D