

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  4:  * This software is proprietary and confidential
5:  5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  6:  */
7:
8:  8:  /*! \class Conversion
9:  9:  class which converts a cv::Mat image from one colorspace to the next colorspace
10: 10: */
11: #include "Conversion.h"
12: namespace Vision {
13: 13:  /*! Constructor of the class      */
14: Conversion::Conversion() {
15:     OriginalColorSpace = None;
16:     ProcessedColorSpace = None;
17: }
18:
19: 19:  /*! Constructor of the class
20: 20:  \param src a cv::Mat object which is the source image
21: 21:  */
22: Conversion::Conversion(const Mat &src) {
23:     OriginalColorSpace = None;
24:     ProcessedColorSpace = None;
25:     OriginalImg = src;
26: }
27:
28: 28:  /*! Copy constructor*/
29: Conversion::Conversion(const Conversion &rhs) {
30:     this->OriginalColorSpace = rhs.OriginalColorSpace;
31:     this->OriginalImg = rhs.OriginalImg;
32:     this->ProcessedColorSpace = rhs.ProcessedColorSpace;
33:     this->ProcessedImg = rhs.ProcessedImg;
34:     this->TempImg = rhs.TempImg;
35: }
36:
37: 37:  /*! De-constructor of the class*/
38: Conversion::~Conversion() {}
39:
40: 40:  /*! Assignment operator*/
41: Conversion &Conversion::operator=(Conversion rhs) {
42:     if (&rhs != this) {
43:         this->OriginalColorSpace = rhs.OriginalColorSpace;
44:         this->OriginalImg = rhs.OriginalImg;
45:         this->ProcessedColorSpace = rhs.ProcessedColorSpace;
46:         this->ProcessedImg = rhs.ProcessedImg;
47:         this->TempImg = rhs.TempImg;
48:     }
49:     return *this;
50: }
51:
52: 52:  /*! Convert the source image from one colorspace to a destination colorspace
53: 53:  - RGB 2 Intensity
54: 54:  - RGB 2 XYZ
55: 55:  - RGB 2 Lab
56: 56:  - RGB 2 Redness Index
57: 57:  - XYZ 2 Lab
58: 58:  - XYZ 2 Redness Index
59: 59:  - Lab 2 Redness Index
60: 60:  \param src a cv::Mat object which is the source image
61: 61:  \param dst a cv::Mat object which is the destination image
62: 62:  \param convertFrom the starting colorspace
63: 63:  \param convertTo the destination colorspace
64: 64:  \param chain use the results from the previous operation default value = false;
65: 65:  */
66: void Conversion::Convert(const Mat &src, Mat &dst, ColorSpace convertFrom,
67:     ColorSpace convertTo, bool chain) {
68:     OriginalImg = src;
69:     Convert(convertFrom, convertTo, chain);
70:     dst = ProcessedImg;
71: }
72:
73: 73:  /*! Convert the source image from one colorspace to a destination colorspace
74: 74:  possibilities are:
75: 75:  - RGB 2 Intensity
76: 76:  - RGB 2 XYZ
77: 77:  - RGB 2 Lab
78: 78:  - RGB 2 Redness Index
79: 79:  - XYZ 2 Lab
80: 80:  - XYZ 2 Redness Index
81: 81:  - Lab 2 Redness Index
82: 82:  \param convertFrom the starting colorspace
83: 83:  \param convertTo the destination colorspace

```

```

84: \param chain use the results from the previous operation default value = false;
85: */
86: void Conversion::Convert(ColorSpace convertFrom, ColorSpace convertTo,
87:                          bool chain) {
88:     OriginalColorSpace = convertFrom;
89:     ProcessedColorSpace = convertTo;
90:
91:     // Exception handling
92:     EMPTY_CHECK(OriginalImg);
93:     currentProg = 0.;
94:     prog_sig(currentProg, "Converting colorspace");
95:
96:     int nData = OriginalImg.rows * OriginalImg.cols;
97:     // uint32_t i, j;
98:
99:     if (convertFrom == RGB && convertTo == Intensity) // RGB 2 Intensity
100:     {
101:         ProcessedImg.create(OriginalImg.size(), CV_8UC1);
102:         uchar *P = ProcessedImg.data;
103:         uchar *O;
104:         CHAIN_PROCESS(chain, O, uchar);
105:
106:         prog_sig(currentProg, "RGB 2 Intensity conversion");
107:         RGB2Intensity(O, P, nData);
108:         currentProg += ProgStep;
109:         prog_sig(currentProg, "RGB 2 Intensity conversion Finished");
110:     } else if (convertFrom == RGB && convertTo == CIE_XYZ) // RGB 2 XYZ
111:     {
112:         ProcessedImg.create(OriginalImg.size(), CV_32FC3);
113:         float *P = (float *)ProcessedImg.data;
114:         uchar *O;
115:         CHAIN_PROCESS(chain, O, uchar);
116:
117:         prog_sig(currentProg, "RGB 2 CIE XYZ conversion");
118:         RGB2XYZ(O, P, nData);
119:         currentProg += ProgStep;
120:         prog_sig(currentProg, "RGB 2 CIE XYZ conversion Finished");
121:     } else if (convertFrom == RGB && convertTo == CIE_lab) // RGB 2 Lab
122:     {
123:         ProcessedImg.create(OriginalImg.size(), CV_32FC3);
124:         float *P = (float *)ProcessedImg.data;
125:         uchar *O;
126:         CHAIN_PROCESS(chain, O, uchar);
127:
128:         prog_sig(currentProg, "RGB 2 CIE XYZ conversion");
129:         RGB2XYZ(O, P, nData);
130:         currentProg += ProgStep;
131:         prog_sig(currentProg, "RGB 2 CIE XYZ conversion Finished");
132:         Convert(CIE_XYZ, CIE_lab, true);
133:     } else if (convertFrom == RGB && convertTo == RI) // RGB 2 RI
134:     {
135:         ProcessedImg.create(OriginalImg.size(), CV_32FC3);
136:         float *P = (float *)ProcessedImg.data;
137:         uchar *O;
138:         CHAIN_PROCESS(chain, O, uchar);
139:
140:         prog_sig(currentProg, "RGB 2 CIE XYZ conversion");
141:         RGB2XYZ(O, P, nData);
142:         currentProg += ProgStep;
143:         prog_sig(currentProg, "RGB 2 CIE XYZ conversion Finished");
144:         Convert(CIE_XYZ, CIE_lab, true);
145:         Convert(CIE_lab, RI, true);
146:     } else if (convertFrom == CIE_XYZ && convertTo == CIE_lab) // XYZ 2 Lab
147:     {
148:         ProcessedImg.create(OriginalImg.size(), CV_32FC3);
149:         float *P = (float *)ProcessedImg.data;
150:         float *O;
151:         CHAIN_PROCESS(chain, O, float);
152:
153:         prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion");
154:         XYZ2Lab(O, P, nData);
155:         currentProg += ProgStep;
156:         prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion Finished");
157:     } else if (convertFrom == CIE_XYZ && convertTo == RI) // XYZ 2 RI
158:     {
159:         ProcessedImg.create(OriginalImg.size(), CV_32FC3);
160:         float *P = (float *)ProcessedImg.data;
161:         float *O;
162:         CHAIN_PROCESS(chain, O, float);
163:
164:         prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion");
165:         XYZ2Lab(O, P, nData);
166:         currentProg += ProgStep;

```

```

167:     prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion Finished");
168:     Convert(CIE_lab, RI, true);
169: } else if (convertFrom == CIE_lab && convertTo == RI) // Lab 2 RI
170: {
171:     ProcessedImg.create(OriginalImg.size(), CV_32FC1);
172:     float *P = (float *)ProcessedImg.data;
173:     float *O;
174:     CHAIN_PROCESS(chain, O, float);
175:
176:     prog_sig(currentProg, "CIE La*b* 2 Redness Index conversion");
177:     Lab2RI(O, P, nData * 3);
178:     currentProg += ProgStep;
179:     prog_sig(currentProg, "CIE La*b* 2 Redness Index conversion Finsihed");
180: } else {
181:     throw Exception::ConversionNotSupportedException();
182: }
183: }
184:
185: /*! Conversion from RGB to Intensity
186: \param O a uchar pointer to the source image
187: \param P a uchar pointer to the destination image
188: \param nData an int indicating the total number of pixels
189: */
190: void Conversion::RGB2Intensity(uchar *O, uchar *P, int nData) {
191:     uint32_t i;
192:     int j;
193:     i = 0;
194:     j = 0;
195:     while (j < nData) {
196:         P[j++] = (*(O + i + 2) * 0.2126 + *(O + i + 1) * 0.7152 +
197:                 *(O + i) * 0.0722); // Grey value
198:         i += 3;
199:     }
200: }
201:
202: /*! Conversion from RGB to CIE XYZ
203: \param O a uchar pointer to the source image
204: \param P a uchar pointer to the destination image
205: \param nData an int indicating the total number of pixels
206: */
207: void Conversion::RGB2XYZ(uchar *O, float *P, int nData) {
208:     uint32_t endData = nData * OriginalImg.step.buf[1];
209:     float R, G, B;
210:     for (uint32_t i = 0; i < endData; i += OriginalImg.step.buf[1]) {
211:         R = static_cast<float>(*(O + i + 2) / 255.0f);
212:         B = static_cast<float>(*(O + i + 1) / 255.0f);
213:         G = static_cast<float>(*(O + i) / 255.0f);
214:         P[i] = (XYZmat[0][0] * R) + (XYZmat[0][1] * B) + (XYZmat[0][2] * G); // X
215:         P[i + 1] = (XYZmat[1][0] * R) + (XYZmat[1][1] * B) + (XYZmat[1][2] * G); // Y
216:         P[i + 2] = (XYZmat[2][0] * R) + (XYZmat[2][1] * B) + (XYZmat[2][2] * G); // Z
217:     }
218: }
219:
220: /*! Conversion from CIE XYZ to CIE La*b*
221: \param O a uchar pointer to the source image
222: \param P a uchar pointer to the destination image
223: \param nData an int indicating the total number of pixels
224: */
225: void Conversion::XYZ2Lab(float *O, float *P, int nData) {
226:     uint32_t endData = nData * 3;
227:     float yy0, xx0, zz0;
228:     for (size_t i = 0; i < endData; i += 3) {
229:         xx0 = *(O + i) / whitePoint[0];
230:         yy0 = *(O + i + 1) / whitePoint[1];
231:         zz0 = *(O + i + 2) / whitePoint[2];
232:
233:         if (yy0 > 0.008856) {
234:             P[i] = (116 * pow(yy0, 0.333f)) - 16; // L
235:         } else {
236:             P[i] = 903.3 * yy0; // L
237:         }
238:
239:         P[i + 1] = 500 * (f_xyz2lab(xx0) - f_xyz2lab(yy0));
240:         P[i + 2] = 200 * (f_xyz2lab(yy0) - f_xyz2lab(zz0));
241:     }
242: }
243:
244: inline float Conversion::f_xyz2lab(float t) {
245:     if (t > 0.008856) {
246:         return pow(t, 0.3333333333f);
247:     }
248:     return 7.787 * t + 0.137931034482759f;
249: }

```

```
250:
251: /*! Conversion from CIE La*b* to Redness Index
252: \param O a uchar pointer to the source image
253: \param P a uchar pointer to the destination image
254: \param nData an int indicating the total number of pixels
255: */
256: void Conversion::Lab2RI(float *O, float *P, int nData) {
257:     uint32_t j = 0;
258:     float L, a, b;
259:     for (int i = 0; i < nData; i += 3) {
260:         L = *(O + i);
261:         a = *(O + i + 1);
262:         b = *(O + i + 2);
263:         P[j++] =
264:             (L * (pow((pow(a, 2.0f) + pow(b, 2.0f)), 0.5f) * (pow(10, 8.2f)))) /
265:             (b * pow(L, 6.0f));
266:     }
267: }
268: }
```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class WrongKernelSizeException
9:  Exception class which is thrown when a wrong kernel size is requested
10: */
11: #pragma once
12:
13: #include <exception>
14: #include <string>
15:
16: using namespace std;
17:
18: namespace Vision {
19: namespace Exception {
20: class WrongKernelSizeException : public std::exception {
21: public:
22:     WrongKernelSizeException(string m = "Wrong kernel dimensions!") : msg(m){};
23:     ~WrongKernelSizeException() _GLIBCXX_USE_NOEXCEPT{};
24:     const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
25:
26: private:
27:     string msg;
28: };
29: }
30: }

```

```
1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class ConversionNotSupportedException
9:  Exception class which is thrown when an illegal conversion is requested.
10: */
11: #pragma once
12:
13: #include <exception>
14: #include <string>
15:
16: using namespace std;
17:
18: namespace Vision {
19: namespace Exception {
20: class ConversionNotSupportedException : public std::exception {
21: public:
22:     ConversionNotSupportedException(
23:         string m = "Requested conversion is not supported!")
24:         : msg(m){};
25:     ~ConversionNotSupportedException() _GLIBCXX_USE_NOEXCEPT{};
26:     const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
27:
28: private:
29:     string msg;
30: };
31: }
32: }
```

```
1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class ChannelMismatchException
9:  Exception class which is thrown when Extracted channel out of bounds exception
10: */
11:
12: #pragma once
13:
14: #include <exception>
15: #include <string>
16:
17: using namespace std;
18:
19: namespace Vision {
20: namespace Exception {
21: class ChannelMismatchException : public std::exception {
22: public:
23:     ChannelMismatchException(
24:         string m = "Extracted channel out of bounds exception!")
25:         : msg(m){};
26:     ~ChannelMismatchException() _GLIBCXX_USE_NOEXCEPT{};
27:     const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
28:
29: private:
30:     string msg;
31: };
32: }
33: }
```

```
1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class Segment
9:  \brief Segmentation algorithms
10: With this class, various segmentation routines can be applied to a greyscale or
11: black and white source image.
12: */
13: #include "Segment.h"
14:
15: namespace Vision {
16:  ///! Constructor of the Segmentation class
17:  Segment::Segment() {}
18:
19:  ///! Constructor of the Segmentation class
20:  Segment::Segment(const Mat &src) {
21:      OriginalImg = src;
22:      ProcessedImg.create(OriginalImg.size(), CV_8UC1);
23:      LabelledImg.create(OriginalImg.size(), CV_16UC1);
24:  }
25:
26:  Segment::Segment(const Segment &rhs) {
27:      this->BlobList = rhs.BlobList;
28:      this->LabelledImg = rhs.LabelledImg;
29:      this->MaxLabel = rhs.MaxLabel;
30:      this->noOfFilteredBlobs = rhs.noOfFilteredBlobs;
31:      this->OriginalImg = rhs.OriginalImg;
32:      this->OriginalImgStats = rhs.OriginalImgStats;
33:      this->ProcessedImg = rhs.ProcessedImg;
34:      this->TempImg = rhs.TempImg;
35:      this->ThresholdLevel = rhs.ThresholdLevel;
36:  }
37:
38:  ///! De-constructor
39:  Segment::~Segment() {}
40:
41:  Segment &Segment::operator=(Segment &rhs) {
42:      if (&rhs != this) {
43:          this->BlobList = rhs.BlobList;
44:          this->LabelledImg = rhs.LabelledImg;
45:          this->MaxLabel = rhs.MaxLabel;
46:          this->noOfFilteredBlobs = rhs.noOfFilteredBlobs;
47:          this->OriginalImg = rhs.OriginalImg;
48:          this->OriginalImgStats = rhs.OriginalImgStats;
49:          this->ProcessedImg = rhs.ProcessedImg;
50:          this->TempImg = rhs.TempImg;
51:          this->ThresholdLevel = rhs.ThresholdLevel;
52:      }
53:      return *this;
54:  }
55:
56:  void Segment::LoadOriginalImg(const Mat &src) {
57:      OriginalImg = src;
58:      ProcessedImg.create(OriginalImg.size(), CV_8UC1);
59:      LabelledImg.create(OriginalImg.size(), CV_16UC1);
60:  }
61:
62:  /*! Determine the threshold level by iteration, between two distribution,
63:  presumably back- and foreground. It works towards the average of the two
64:  averages and finally sets the threshold with two time the standard deviation
65:  from the mean of the set object
66:  \param TypeObject is an enumerator indicating if the bright or the dark pixels
67:  are the object and should be set to one
68:  \return The threshold level as an uint8_t */
69:  uint8_t Segment::GetThresholdLevel(TypeOfObjects TypeObject) {
70:      // Exception handling
71:      EMPTY_CHECK(OriginalImg);
72:      CV_Assert(OriginalImg.depth() != sizeof(uchar));
73:
74:      // Calculate the statistics of the whole picture
75:      ucharStat_t OriginalImgStats(OriginalImg.data, OriginalImg.rows,
76:                                   OriginalImg.cols);
77:
78:      // Sets the initial threshold with the mean of the total picture
79:      pair<uchar, uchar> T;
80:      T.first = (uchar)(OriginalImgStats.Mean + 0.5);
81:      T.second = 0;
82:
83:      uchar Rstd = 0;
```



```
84:  uchar Lstd = 0;
85:  uchar Rmean = 0;
86:  uchar Lmean = 0;
87:
88:  // Iterate till optimum Threshold is found between back- & foreground
89:  while (T.first != T.second) {
90:      // Gets an array of the left part of the histogram
91:      uint32_t i = T.first;
92:      uint32_t *Left = new uint32_t[i]{};
93:      while (i-- > 0) {
94:          Left[i] = OriginalImgStats.bins[i];
95:      }
96:
97:      // Gets an array of the right part of the histogram
98:      uint32_t rightEnd = 256 - T.first;
99:      uint32_t *Right = new uint32_t[rightEnd]{};
100:      i = rightEnd;
101:      while (i-- > 0) {
102:          Right[i] = OriginalImgStats.bins[i + T.first];
103:      }
104:
105:      // Calculate the statistics of both histograms,
106:      // taking into account the current threshold
107:      ucharStat_t sLeft(Left, 0, T.first);
108:      ucharStat_t sRight(Right, T.first, 256);
109:
110:      // Calculate the new threshold the mean of the means
111:      T.second = T.first;
112:      T.first = (uchar)((sLeft.Mean + sRight.Mean) / 2) + 0.5;
113:
114:      Rmean = (uchar)(sRight.Mean + 0.5);
115:      Lmean = (uchar)(sLeft.Mean + 0.5);
116:      Rstd = (uchar)(sRight.Std + 0.5);
117:      Lstd = (uchar)(sLeft.Std + 0.5);
118:      delete[] Left;
119:      delete[] Right;
120:  }
121:
122:  // Assumes the pixel value of the sought object lies between 2 sigma
123:  int val = 0;
124:  switch (TypeObject) {
125:      case Bright:
126:          val = Rmean - (sigma * Rstd) - thresholdOffset;
127:          if (val < 0) {
128:              val = 0;
129:          } else if (val > 255) {
130:              val = 255;
131:          }
132:          T.first = (uchar)val;
133:          break;
134:      case Dark:
135:          val = Lmean + (sigma * Lstd) + thresholdOffset;
136:          if (val < 0) {
137:              val = 0;
138:          } else if (val > 255) {
139:              val = 255;
140:          }
141:          T.first = (uchar)val;
142:          break;
143:  }
144:
145:  return T.first;
146: }
147:
148: /*! Convert a greyscale image to a BW using an automatic Threshold
149: \param src is the source image as a cv::Mat
150: \param dst destination image as a cv::Mat
151: \param TypeObject is an enumerator indicating if the bright or the dark pixels
152: are the object and should be set to one */
153: void Segment::ConvertToBW(const Mat &src, Mat &dst, TypeOfObjects Typeobjects) {
154:     OriginalImg = src;
155:     ProcessedImg.create(OriginalImg.size(), CV_8UC1);
156:     LabelledImg.create(OriginalImg.size(), CV_16UC1);
157:     ConvertToBW(Typeobjects);
158:     dst = ProcessedImg;
159: }
160:
161: /*! Convert a greyscale image to a BW using an automatic Threshold
162: \param TypeObject is an enumerator indicating if the bright or the dark pixels
163: are the object and should be set to one */
164: void Segment::ConvertToBW(TypeOfObjects Typeobjects) {
165:     // Determine the threshold
166:     uchar T = GetThresholdLevel(Typeobjects);
```

```
167:
168:     // Threshold the picture
169:     Threshold(T, Typeobjects);
170: }
171:
172: /*! Convert a greyscale image to a BW
173: \param t uchar set the value which is the tipping point
174: \param TypeObject is an enumerator indicating if the bright or the dark pixels
175: are the object and should be set to one */
176: void Segment::Threshold(uchar t, TypeOfObjects Typeobjects) {
177:     // Exception handling
178:     EMPTY_CHECK(OriginalImg);
179:     CV_Assert(OriginalImg.depth() != sizeof(uchar) ||
180:               OriginalImg.depth() != sizeof(uint16_t));
181:
182:     // Create LUT
183:     uchar LUT_newValue[256]{0};
184:     if (Typeobjects == Bright) {
185:         for (uint32_t i = t; i < 256; i++) {
186:             LUT_newValue[i] = 1;
187:         }
188:     } else {
189:         for (uint32_t i = 0; i <= t; i++) {
190:             LUT_newValue[i] = 1;
191:         }
192:     }
193:
194:     // Create the pointers to the data
195:     uchar *P = ProcessedImg.data;
196:     uchar *O = OriginalImg.data;
197:
198:     // Fills the ProcessedImg with either a 0 or 1
199:     for (int i = 0; i < OriginalImg.cols * OriginalImg.rows; i++) {
200:         P[i] = LUT_newValue[O[i]];
201:     }
202: }
203:
204: /*! Set all the border pixels to a set value
205: \param *P uchar pointer to the Mat.data
206: \param setValue uchar the value which is written to the border pixels
207: */
208: void Segment::SetBorder(uchar *P, uchar setValue) {
209:     // Exception handling
210:     EMPTY_CHECK(OriginalImg);
211:     CV_Assert(OriginalImg.depth() != sizeof(uchar) ||
212:               OriginalImg.depth() != sizeof(uint16_t));
213:
214:     uint32_t nData = OriginalImg.cols * OriginalImg.rows;
215:
216:     // Set borderPixels to 2
217:     uint32_t i = 0;
218:     uint32_t pEnd = OriginalImg.cols + 1;
219:
220:     // Set the top row to value 2
221:     while (i < pEnd) {
222:         P[i++] = setValue;
223:     }
224:
225:     // Set the bottom row to value 2
226:     i = nData + 1;
227:     pEnd = nData - OriginalImg.cols;
228:     while (i-- > pEnd) {
229:         P[i] = setValue;
230:     }
231:
232:     // Sets the first and the last Column to 2
233:     i = 1;
234:     pEnd = OriginalImg.rows;
235:     while (i < pEnd) {
236:         P[(i * OriginalImg.cols) - 1] = setValue;
237:         P[(i++ * OriginalImg.cols)] = setValue;
238:     }
239: }
240:
241: /*! Remove the blobs that are connected to the border
242: \param conn set the pixel connection eight or four
243: \param chain use the results from the previous operation default value = false;
244: */
245: void Segment::RemoveBorderBlobs(uint32_t border, bool chain) {
246:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
247:     EMPTY_CHECK(OriginalImg);
248:     // make Pointers
249:     uchar *O;
```

```

250: CHAIN_PROCESS(chain, 0, uchar);
251: if (chain) {
252:     ProcessedImg = TempImg.clone();
253: } else {
254:     ProcessedImg = OriginalImg.clone();
255: }
256:
257: SHOW_DEBUG_IMG(OriginalImg, uchar, 255, "Original Image RemoverBorderBlobs!",
258:     true);
259: SHOW_DEBUG_IMG(TempImg, uchar, 255, "Temp Image RemoverBorderBlobs!", true);
260:
261: uchar *P = ProcessedImg.data;
262: uint32_t cols = ProcessedImg.cols;
263: uint32_t rows = ProcessedImg.rows;
264:
265: try {
266:     for (uint32_t i = 0; i < border; i++) {
267:         for (uint32_t j = 0; j < cols; j++) {
268:             if (O[(i * cols) + j] == 1 && P[(i * cols) + j] != 2) {
269:                 cv::floodFill(ProcessedImg, cv::Point(j, i), (uchar)2);
270:             }
271:         }
272:     }
273:
274:     for (uint32_t i = rows - border - 1; i < rows; i++) {
275:         for (uint32_t j = 0; j < cols; j++) {
276:             if (O[(i * cols) + j] == 1 && P[(i * cols) + j] != 2) {
277:                 cv::floodFill(ProcessedImg, cv::Point(j, i), (uchar)2);
278:             }
279:         }
280:     }
281:
282:     for (uint32_t i = border; i < rows - border; i++) {
283:         for (uint32_t j = 0; j < border; j++) {
284:             if (O[(i * cols) + j] == 1 && P[(i * cols) + j] != 2) {
285:                 cv::floodFill(ProcessedImg, cv::Point(j, i), (uchar)2);
286:             }
287:             if (O[(i * cols) + (cols - j - 1)] == 1 &&
288:                 P[(i * cols) + (cols - j - 1)] != 2) {
289:                 cv::floodFill(ProcessedImg, cv::Point(cols - j - 1, i), (uchar)2);
290:             }
291:         }
292:     }
293: } catch (cv::Exception &e) {
294: }
295: SHOW_DEBUG_IMG(ProcessedImg, uchar, 255,
296:     "Processed Image RemoverBorderBlobs before LUT!", true);
297:
298: // Change values 2 -> 0
299: uchar LUT_newValue[3]{0, 1, 0};
300: P = ProcessedImg.data;
301: uint32_t nData = rows * cols;
302: for (uint32_t i = 0; i < nData; i++) {
303:     P[i] = LUT_newValue[P[i]];
304: }
305:
306: SHOW_DEBUG_IMG(ProcessedImg, uchar, 255,
307:     "Processed Image RemoverBorderBlobs!", true);
308: }
309:
310: /*! Label all the individual blobs in a BW source image. The result are written
311: to the labelledImg as an ushort
312: \param conn set the pixel connection eight or four
313: \param chain use the results from the previous operation default value = false;
314: \param minBlobArea minimum area when an artifact is considered a blob
315: */
316: void Segment::LabelBlobs(bool chain, uint16_t minBlobArea, Connected conn) {
317:     // Exception handling
318:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
319:     EMPTY_CHECK(OriginalImg);
320:
321:     // make the Pointers to the data
322:     uchar *O;
323:     if (chain) {
324:         TempImg = ProcessedImg.clone();
325:         ProcessedImg = cv::Mat(OriginalImg.rows, OriginalImg.cols, CV_16UC1);
326:         O = (uchar *)TempImg.data;
327:     } else {
328:         O = (uchar *)OriginalImg.data;
329:     }
330:     uint16_t *P = (uint16_t *)LabelledImg.data;
331:
332:     uint32_t nCols = OriginalImg.cols;

```

```
333:   uint32_t nRows = OriginalImg.rows;
334:   uint32_t nData = nCols * nRows;
335:
336:   vector<vector<uint16_t>> CLdownstream;
337:
338:   ConnectedBlobs(0, P, CLdownstream, nCols, nRows,
339:                 conn); // First loop through the image
340:   SortAdjacencyList(
341:     CLdownstream); // Sort all the adjacencylists and make unique,
342:
343:   // identify all the lowest values in the adjacent list
344:   uint16_t *valueArr = new uint16_t[CLdownstream.size()];
345:   for (int i = CLdownstream.size() - 1; i >= 0; --i) {
346:     std::vector<uint16_t *> route;
347:     uint16_t minVal = i;
348:
349:     for (uint32_t j = 0; j < CLdownstream[i].size(); j++) {
350:
351:       // add the first node to the queue;
352:       route.push_back(&CLdownstream[i][j]);
353:
354:       // iterate till the last node
355:       bool lastNodeReached = false;
356:       while (!lastNodeReached) {
357:         uint32_t nodesVisited = route.size() - 1;
358:         if (*route[nodesVisited] < minVal) {
359:           minVal = *route[nodesVisited];
360:         }
361:         route.push_back(&CLdownstream[*route[nodesVisited]][0]);
362:         if (route[nodesVisited] == route[nodesVisited + 1]) {
363:           route.pop_back();
364:           lastNodeReached = true;
365:         }
366:       }
367:       // Set all values to the lowest value
368:       for (uint32_t k = 0; k < route.size(); k++) {
369:         *route[k] = minVal;
370:       }
371:     }
372:     valueArr[i] = minVal;
373:   }
374:
375:   // Make numbers consecutive
376:   MakeConsecutive(valueArr, CLdownstream.size(), MaxLabel);
377:
378:   // Second loop through the pixels to give the values a final value
379:   for_each(P, P + nData, [&](uint16_t &V) { V = valueArr[V]; });
380:   delete[] valueArr;
381: }
382:
383: /*! Create a BW image with only edges from a BW image
384: \param src source image as a const cv::Mat
385: \param dst destination image as a cv::Mat
386: \param conn set the pixel connection eight or four
387: \param chain use the results from the previous operation default value = false;
388: */
389: void Segment::GetEdges(const Mat &src, Mat &dst, bool chain, Connected conn) {
390:   OriginalImg = src;
391:   GetEdges(chain, conn);
392:   dst = ProcessedImg;
393: }
394:
395: /*! Create a BW image with only edges from a BW image
396: \param conn set the pixel connection eight or four
397: \param chain use the results from the previous operation default value = false;
398: */
399: void Segment::GetEdges(bool chain, Connected conn) {
400:   // Exception handling
401:   CV_Assert(OriginalImg.depth() != sizeof(uchar));
402:   EMPTY_CHECK(OriginalImg);
403:
404:   // make Pointers
405:   uchar *O;
406:   CHAIN_PROCESS(chain, O, uchar);
407:   uchar *P = ProcessedImg.data;
408:
409:   uint32_t nCols = OriginalImg.cols;
410:   uint32_t nRows = OriginalImg.rows;
411:   uint32_t nData = nCols * nRows;
412:   uint32_t pEnd = nData + 1;
413:   uint32_t i = 0;
414:
415:   // Loop through the image and set each pixel which has a zero neighbor set it
```

```
416: // to two.
417: if (conn == Four) {
418:     // Loop through the picture
419:     while (i < pEnd) {
420:         // If current value = zero processed value = zero
421:         if (O[i] == 0) {
422:             P[i] = 0;
423:         }
424:         // If current value = 1 check North West, South and East and act
425:         // accordingly
426:         else if (O[i] == 1) {
427:             uchar *nPixels = new uchar[4];
428:             nPixels[0] = O[i - 1];
429:             nPixels[1] = O[i - nCols];
430:             nPixels[2] = O[i + 1];
431:             nPixels[3] = O[i + nCols];
432:
433:             // Sort the neighbors for easier checking
434:             SoilMath::Sort::QuickSort<uchar>(nPixels, 4);
435:             if (nPixels[0] == 0) {
436:                 P[i] = 1;
437:             } else {
438:                 P[i] = 0;
439:             }
440:         } else {
441:             throw Exception::PixelValueOutOfBoundsException();
442:         }
443:         i++;
444:     }
445: } else {
446:     // Loop through the picture
447:     while (i < pEnd) {
448:         // If current value = zero processed value = zero
449:         if (O[i] == 0) {
450:             P[i] = 0;
451:         }
452:         // If current value = 1 check North West, South and East and act
453:         // accordingly
454:         else if (O[i] == 1) {
455:             uchar *nPixels = new uchar[8];
456:             nPixels[0] = O[i - 1];
457:             nPixels[1] = O[i - nCols];
458:             nPixels[2] = O[i - nCols - 1];
459:             nPixels[3] = O[i - nCols + 1];
460:             nPixels[4] = O[i + 1];
461:             nPixels[5] = O[i + nCols + 1];
462:             nPixels[6] = O[i + nCols];
463:             nPixels[7] = O[i + nCols - 1];
464:
465:             // Sort the neighbors for easier checking
466:             SoilMath::Sort::QuickSort<uchar>(nPixels, 8);
467:
468:             if (nPixels[0] == 0) {
469:                 P[i] = 1;
470:             } else {
471:                 P[i] = 0;
472:             }
473:         } else {
474:             throw Exception::PixelValueOutOfBoundsException();
475:         }
476:         i++;
477:     }
478: }
479: }
480:
481: void Segment::GetEdgesEroding(bool chain) {
482:     // Exception handling
483:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
484:     EMPTY_CHECK(OriginalImg);
485:
486:     // make Pointers
487:     uchar *O;
488:     CHAIN_PROCESS(chain, O, uchar);
489:     uchar *P = ProcessedImg.data;
490:
491:     uint32_t nCols = OriginalImg.cols;
492:     uint32_t nRows = OriginalImg.rows;
493:     uint32_t nData = nCols * nRows;
494:
495:     // Setup the erosion
496:     MorphologicalFilter eroder;
497:     if (chain) {
498:         eroder.OriginalImg = TempImg;
```

```
499:     } else {
500:         eroder.OriginalImg = OriginalImg;
501:     }
502:     // Setup the processed image of the eroder
503:     eroder.ProcessedImg.create(OriginalImg.size(), CV_8UC1);
504:     eroder.ProcessedImg.setTo(0);
505:     // Setup the mask
506:     Mat mask(3, 3, CV_8UC1, 1);
507:     // Erode the image
508:     eroder.Erosion(mask, false);
509:
510:     // Loop through the image and set the not eroded pixels to zero
511:     for (uint32_t i = 0; i < nData; i++) {
512:         if (O[i] != eroder.ProcessedImg.data[i]) {
513:             P[i] = 1;
514:         } else {
515:             P[i] = 0;
516:         }
517:     }
518:
519:     // ProcessedImg = OriginalImg.clone() - eroder.ProcessedImg.clone();
520:
521:     SHOW_DEBUG_IMG(eroder.ProcessedImg, uchar, 255, "Eroded img Processed Image!",
522:         true);
523:     SHOW_DEBUG_IMG(ProcessedImg, uchar, 255, "GetEdgesEroding Processed Image!",
524:         true);
525: }
526:
527: /*! Create a BlobList subtracting each individual blob out of a Labelled image.
528: If the labelled image is empty build a new one with a BW image.
529: \param conn set the pixel connection eight or four
530: \param chain use the results from the previous operation default value = false;
531: */
532: void Segment::GetBlobList(bool chain, Connected conn) {
533:     // Exception handling
534:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
535:     EMPTY_CHECK(OriginalImg);
536:
537:     // If there isn't a labelledImg make one
538:     if (MaxLabel < 1) {
539:         LabelBlobs(chain, 5, conn);
540:     }
541:
542:     // Make an empty BlobList
543:     uint32_t nCols = OriginalImg.cols;
544:     uint32_t nRows = OriginalImg.rows;
545:     uint32_t nData = nCols * nRows;
546:     RectList_t rectList;
547:
548:     // Calculate Stats the statistics
549:     uint16Stat_t LabelStats((uint16_t *)LabelledImg.data, LabelledImg.cols,
550:         LabelledImg.rows, MaxLabel + 1, 0, MaxLabel);
551:
552:     BlobList.reserve(LabelStats.EndBin);
553:     rectList.reserve(LabelStats.EndBin);
554:
555:     BlobList.push_back(Blob_t(0, 0));
556:     rectList.push_back(Rect_t(0, 0, 0, 0));
557:
558:     for (uint32_t i = 1; i < LabelStats.EndBin; i++) {
559:         BlobList.push_back(Blob_t(i, LabelStats.bins[i]));
560:         rectList.push_back(Rect_t(nCols, nRows, 0, 0));
561:     }
562:
563:     // make Pointers
564:     uint16_t *L = (uint16_t *)LabelledImg.data;
565:
566:     uint32_t currentX, currentY;
567:     // uint16_t leftX, leftY, rightX, rightY;
568:     // Loop through the labeled image and extract the Blobs
569:     for (uint32_t i = 0; i < nData; i++) {
570:         if (L[i] != 0) {
571:             /* Determine the current x and y value of the current blob and
572:             checks if it is min/max */
573:             currentY = i / nCols;
574:             currentX = i % nCols;
575:
576:             // Min value
577:             if (currentX < rectList[L[i]].leftX) {
578:                 rectList[L[i]].leftX = currentX;
579:             }
580:             if (currentY < rectList[L[i]].leftY) {
581:                 rectList[L[i]].leftY = currentY;
```

```
582:     }
583:
584:     // Max value
585:     if (currentX > rectList[L[i]].rightX) {
586:         rectList[L[i]].rightX = currentX;
587:     }
588:     if (currentY > rectList[L[i]].rightY) {
589:         rectList[L[i]].rightY = currentY;
590:     }
591: }
592: }
593:
594: // Loop through the BlobList and finalize it
595: uint8_t *LUT_filter = new uint8_t[MaxLabel + 1]{};
596: for (uint32_t i = 1; i <= MaxLabel; i++) {
597:     LUT_filter[i] = 1;
598:     BlobList[i].ROI.y = rectList[i].leftY;
599:     BlobList[i].ROI.x = rectList[i].leftX;
600:     BlobList[i].ROI.height = rectList[i].rightY - rectList[i].leftY + 1;
601:     BlobList[i].ROI.width = rectList[i].rightX - rectList[i].leftX + 1;
602:     BlobList[i].Img = CopyMat<uint8_t, uint16_t>(
603:         LabelledImg(BlobList[i].ROI).clone(), LUT_filter, CV_8UC1);
604:     //SHOW_DEBUG_IMG(BlobList[i].Img, uchar, 255, "Blob", true);
605:     LUT_filter[i] = 0;
606: }
607: delete[] LUT_filter;
608:
609: // Remove background blob
610: BlobList.erase(BlobList.begin());
611: }
612:
613: void Segment::FillHoles(bool chain) {
614:     // Exception handling
615:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
616:     EMPTY_CHECK(OriginalImg);
617:
618:     // make Pointers
619:     uchar *O;
620:     CHAIN_PROCESS(chain, O, uchar);
621:     if (chain) {
622:         ProcessedImg = TempImg.clone();
623:     } else {
624:         ProcessedImg = OriginalImg.clone();
625:     }
626:
627:     uchar *P = ProcessedImg.data;
628:
629:     // Determine the starting point of the floodfill
630:     int itt = -1;
631:     while (P[++itt] != 0)
632:         ;
633:     uint16_t row = static_cast<uint16_t>(itt / OriginalImg.rows);
634:     uint16_t col = static_cast<uint16_t>(itt % OriginalImg.rows);
635:
636:     // Fill the outside
637:     // FloodFill(0, P, row, col, 2, 0);
638:
639:     try {
640:         cv::floodFill(ProcessedImg, cv::Point(col, row), cv::Scalar(2));
641:     } catch (cv::Exception &e) {
642:     }
643:
644:     // Set the unreached areas to 1 and the outside to 0;
645:     uchar LUT_newVal[3] = {1, 1, 0};
646:     uint32_t nData = OriginalImg.rows * OriginalImg.cols;
647:     uint32_t i = 0;
648:     while (i <= nData) {
649:         P[i] = LUT_newVal[P[i]];
650:         i++;
651:     }
652: }
653:
654: /*!
655:  * \brief Segment::FloodFill
656:  * \param O
657:  * \param P
658:  * \param row
659:  * \param col
660:  * \param fillValue
661:  * \param OldValue
662:  */
663: void Segment::FloodFill(uchar *O, uchar *P, uint16_t row, uint16_t col,
664:     uchar fillValue, uchar OldValue) {
```

```

665:     if (row < 0 || row > OriginalImg.rows) {
666:         return;
667:     }
668:     if (col < 0 || col > OriginalImg.cols) {
669:         return;
670:     }
671:     if (P[col + row * OriginalImg.rows] == OldValue) {
672:         P[col + row * OriginalImg.rows] = fillValue;
673:         FloodFill(O, P, row + 1, col, fillValue, OldValue);
674:         FloodFill(O, P, row, col + 1, fillValue, OldValue);
675:         FloodFill(O, P, row - 1, col, fillValue, OldValue);
676:         FloodFill(O, P, row, col - 1, fillValue, OldValue);
677:     }
678: }
679:
680: /*!
681:  * \brief Segment::SortAdjacencyList Sort the the sub vectors
682:  * \param adj std::vector<std::vector<uint16_t>> &adj
683:  */
684: void Segment::SortAdjacencyList(std::vector<std::vector<uint16_t>> &adj) {
685:     uint32_t j = 0;
686:     for_each(adj.begin(), adj.end(), [&](std::vector<uint16_t> &L) {
687:         std::sort(L.begin(), L.end());
688:         std::vector<uint16_t>::iterator it;
689:         it = std::unique(L.begin(), L.end());
690:         L.resize(std::distance(L.begin(), it));
691:         if (L.size() > 1) {
692:             for (std::vector<uint16_t>::iterator iter = L.begin(); iter != L.end();
693:                  ++iter) {
694:                 if (*iter == j) {
695:                     L.erase(iter);
696:                     break;
697:                 }
698:             }
699:         }
700:         j++;
701:     });
702: }
703:
704: /*!
705:  * \brief Segment::ConnectedBlobs Connect all the blobs and created the
706:  * adjacency list
707:  * \param O
708:  * \param P
709:  * \param adj
710:  * \param nCols
711:  * \param nRows
712:  * \param conn
713:  */
714: void Segment::ConnectedBlobs(uchar *O, uint16_t *P,
715:                               std::vector<std::vector<uint16_t>> &adj,
716:                               uint32_t nCols, uint32_t nRows, Connected conn) {
717:     // Determine the size of the array for beginning and endrow and middle of a
718:     // row
719:     uint32_t noConn[3] = {static_cast<uint32_t>(conn),
720:                           (static_cast<uint32_t>(conn) / 2),
721:                           (static_cast<uint32_t>(conn) / 2) + 1};
722:     uint32_t lastConn[3] = {noConn[0] - 1, noConn[1] - 1, noConn[2] - 1};
723:     uint32_t nData = nCols * nRows;
724:
725:     uint16_t currentlbl = 0;
726:     vector<uint16_t> zeroVector;
727:     zeroVector.push_back(currentlbl);
728:     adj.push_back(zeroVector);
729:
730:     // Determine which borderpixels should be handled differently
731:     uchar *nRow = new uchar[nData]{};
732:     for (uint32_t i = nCols; i < nData; i += nCols) {
733:         nRow[i] = 1;
734:         nRow[i - 1] = 2;
735:     }
736:
737:     // Set the first pixel
738:     if (O[0] == 0) {
739:         P[0] = 0;
740:     } else if (O[0] == 1) {
741:         P[0] = 1;
742:     } else {
743:         throw Exception::PixelValueOutOfBoundException();
744:     }
745:
746:     // Walk through the toprow and determine if it's a new blob or it's connected
747:     // with previously determine blob

```



```

748:   for (uint32_t i = 1; i < nCols; i++) {
749:       if (O[i] == 0) {
750:           P[i] = 0;
751:       } else if (O[i] == 1) {
752:           // If West is zero assume this is a new blob
753:           if (P[i - 1] == 0) {
754:               P[i] = ++currentlbl;
755:               vector<uint16_t> cVector;
756:               cVector.push_back(currentlbl);
757:               adj.push_back(cVector);
758:           } else { // set as previous blob
759:               P[i] = P[i - 1];
760:           }
761:       } else { // Value of of bounds
762:           throw Exception::PixelValueOutOfBounds();
763:       }
764:   }
765:
766:   // walk through each pixel and determine if it's a new blob or it's connected
767:   // with previously determine blob
768:   for (uint32_t i = OriginalImg.cols; i < nData; i++) {
769:       if (O[i] == 0) { // Original pixel = 0
770:           P[i] = 0;
771:       } else if (O[i] == 1) {
772:           // Get an array of Neighboring Pixels
773:           uint16_t *nPixels = new uint16_t[noConn[nRow[i]]];
774:           if (nRow[i] != 1) {
775:               nPixels[0] = P[i - 1];
776:           }
777:           uint32_t j = i - nCols - ((nRow[i] == 1) ? 0 : ((conn == Four) ? 0 : 1));
778:           for_each(nPixels + ((nRow[i] != 1) ? 1 : 0), nPixels + noConn[nRow[i]],
779:               [&](uint16_t &N) { N = P[j++]; });
780:
781:           // Sort the neighbors for easier checking
782:           SoilMath::Sort::QuickSort<uint16_t>(nPixels, noConn[nRow[i]]);
783:
784:           // If all are zero assume this is a new blob
785:           if (nPixels[lastConn[nRow[i]]] == 0) {
786:               P[i] = ++currentlbl;
787:               vector<uint16_t> cVector;
788:               cVector.push_back(currentlbl);
789:               adj.push_back(cVector);
790:           } else {
791:               /* Sets the processed value to the smallest non-zero value and update
792:               * the connectedLabels */
793:               for (uint32_t j = 0; j < noConn[nRow[i]]; j++) {
794:                   if (nPixels[j] > 0) {
795:                       P[i] = nPixels[j];
796:                       break;
797:                   }
798:               }
799:
800:               /* If previous blobs belong to different connected components set the
801:               * current processed value to the lowest value and remember that the
802:               * other values should be the lowest value*/
803:               if (P[i] != nPixels[lastConn[nRow[i]]]) {
804:                   for (int j = lastConn[nRow[i]]; j >= 0; --j) {
805:                       if (nPixels[j] <= P[i]) {
806:                           break;
807:                       } else {
808:                           adj[nPixels[j]].push_back(P[i]);
809:                       }
810:                   }
811:               }
812:           }
813:           delete[] nPixels;
814:       } else {
815:           throw Exception::PixelValueOutOfBounds();
816:       }
817:   }
818:   delete[] nRow;
819: }
820:
821: /*!
822: * \brief Segment::InvertAdjacencyList invert the adjecencylist for upstream
823: * (unused)
824: * \param adj
825: * \param adjInv
826: */
827: void Segment::InvertAdjacencyList(std::vector<std::vector<uint16_t>> &adj,
828:     std::vector<std::vector<uint16_t>> &adjInv) {
829:     // Build the inverted vector
830:     adjInv.resize(adj.size());

```

```
831:  uint16_t count = 0;
832:  for_each(adj.begin(), adj.end(), [&](std::vector<uint16_t> &V) {
833:      for_each(V.begin(), V.end(),
834:          [&](uint16_t &C) { adjInv[C].push_back(count); });
835:      count++;
836:  });
837: }
838:
839: /*!
840: * \brief Segment::MakeConsecutive make the valueArr consecutive numbers
841: * \param valueArr
842: * \param noElem
843: * \param maxLabel
844: */
845: void Segment::MakeConsecutive(uint16_t *valueArr, uint32_t noElem,
846:     uint16_t &maxLabel) {
847:     std::vector<std::vector<uint16_t>> conseq;
848:     conseq.resize(noElem);
849:     for (uint32_t i = 0; i < noElem; i++) {
850:         conseq[valueArr[i]].push_back(i);
851:     }
852:     uint32_t count = 1;
853:     for (uint32_t i = 1; i < noElem; i++) {
854:         if (conseq[i].size() > 0) {
855:             for (uint32_t j = 0; j < conseq[i].size(); j++) {
856:                 valueArr[conseq[i][j]] = count;
857:             }
858:             count++;
859:         }
860:     }
861:     maxLabel = count - 1;
862: }
863:
864: /*!
865: * \brief Segment::MakeConsecutive probably a fault in this function. Don't use
866: * \param valueArr
867: * \param keyArr
868: * \param noElem
869: * \param maxlabel
870: */
871: void Segment::MakeConsecutive(uint16_t *valueArr, uint16_t *keyArr,
872:     uint16_t noElem, uint16_t &maxlabel) {
873:     SoilMath::Sort::QuickSort<uint16_t>(valueArr, keyArr, noElem);
874:     uint16_t count = 0;
875:     for (uint32_t i = 1; i < noElem; i++) {
876:         if (valueArr[i] != valueArr[i - 1]) {
877:             count++;
878:         }
879:         valueArr[i] = count;
880:     }
881:     SoilMath::Sort::QuickSort<uint16_t>(keyArr, valueArr, noElem);
882:     delete[] keyArr;
883:     maxlabel = count;
884: }
885: }
```

```
1: /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8: /*! Collection header of all the basic Vision headers*/
9:
10: #pragma once
11: #include "Conversion.h"
12: #include "Enhance.h"
13: #include "Segment.h"
14: #include "MorphologicalFilter.h"
```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  #pragma once
9:  // Debuging helper macros
10: #ifndef DEBUG
11: #define DEBUG
12: #endif
13:
14: #ifdef DEBUG
15: #include <limits>
16: #include <opencv2/highgui/highgui.hpp>
17: #include <vector>
18: #include "ImageProcessing.h"
19: #ifndef SHOW_DEBUG_IMG
20: #define SHOW_DEBUG_IMG(img, Tl, maxVal, windowName, scale) \
21:     Vision::ImageProcessing::ShowDebugImg<Tl>(img, maxVal, windowName, scale)
22: #endif // !SHOW_DEBUG_IMG
23: #else
24: #ifndef SHOW_DEBUG_IMG
25: #define SHOW_DEBUG_IMG(img, Tl, maxVal, windowName, scale)
26: #endif // !SHOW_DEBUG_IMG
27: #endif

```

```
1: /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8: /*! \class PixelValueOutOfBoundsException
9:  Exception class which is thrown when an unexpected pixel value has to be
10: computed
11: */
12: #pragma once
13:
14: #include <exception>
15: #include <string>
16:
17: using namespace std;
18:
19: namespace Vision {
20: namespace Exception {
21: class PixelValueOutOfBoundsException : public std::exception {
22: public:
23:     PixelValueOutOfBoundsException(string m = "Current pixel value out of bounds!")
24:         : msg(m){};
25:     ~PixelFormatValueOutOfBoundsException() _GLIBCXX_USE_NOEXCEPT{};
26:     const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
27:
28: private:
29:     string msg;
30: };
31: }
32: }
```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class EmptyImageException
9:  Exception class which is thrown when operations are about to start on a empty
10: image.
11: */
12:
13: #pragma once
14:
15: #include <exception>
16: #include <string>
17:
18: using namespace std;
19:
20: namespace Vision {
21: namespace Exception {
22: class EmptyImageException : public std::exception {
23: public:
24:     EmptyImageException(string m = "Empty Image!") : msg(m){};
25:     ~EmptyImageException() _GLIBCXX_USE_NOEXCEPT{};
26:     const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
27:
28: private:
29:     string msg;
30: };
31: }
32: }

```

```
1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  #pragma once
9:  #define MORPHOLOGICALFILTER_VERSION 1
10:
11:  #include "ImageProcessing.h"
12:
13:  namespace Vision {
14:  class MorphologicalFilter : public ImageProcessing {
15:  public:
16:      enum FilterType { OPEN, CLOSE, ERODE, DILATE, NONE };
17:
18:      MorphologicalFilter();
19:      MorphologicalFilter(FilterType filtertype);
20:      MorphologicalFilter(const Mat &src, FilterType filtertype = FilterType::NONE);
21:      MorphologicalFilter(const MorphologicalFilter &rhs);
22:
23:      ~MorphologicalFilter();
24:
25:      MorphologicalFilter &operator=(MorphologicalFilter &rhs);
26:
27:      void Dilation(const Mat &mask, bool chain = false);
28:      void Erosion(const Mat &mask, bool chain = false);
29:
30:      void Close(const Mat &mask, bool chain = false);
31:      void Open(const Mat &mask, bool chain = false);
32:
33:  private:
34:      void Filter(const Mat &mask, bool chain, uchar startVal, uchar newVal,
35:                  uchar switchVal);
36:  };
37: }
```

```
1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  #include "MorphologicalFilter.h"
9:
10: namespace Vision {
11: MorphologicalFilter::MorphologicalFilter() {}
12:
13: MorphologicalFilter::MorphologicalFilter(FilterType filtertype) {
14:     switch (filtertype) {
15:     case FilterType::OPEN:
16:         Open(OriginalImg);
17:         break;
18:     case FilterType::CLOSE:
19:         Close(OriginalImg);
20:         break;
21:     case FilterType::ERODE:
22:         Erosion(OriginalImg);
23:         break;
24:     case FilterType::DILATE:
25:         Dilation(OriginalImg);
26:         break;
27:     case FilterType::NONE:
28:         break;
29:     }
30: }
31:
32: MorphologicalFilter::MorphologicalFilter(const Mat &src,
33:                                         FilterType filtertype) {
34:     OriginalImg = src;
35:     ProcessedImg.create(OriginalImg.size(), CV_8UC1);
36:     switch (filtertype) {
37:     case FilterType::OPEN:
38:         Open(OriginalImg);
39:         break;
40:     case FilterType::CLOSE:
41:         Close(OriginalImg);
42:         break;
43:     case FilterType::ERODE:
44:         Erosion(OriginalImg);
45:         break;
46:     case FilterType::DILATE:
47:         Dilation(OriginalImg);
48:         break;
49:     case FilterType::NONE:
50:         break;
51:     }
52: }
53:
54: MorphologicalFilter::MorphologicalFilter(const MorphologicalFilter &rhs) {
55:     this->OriginalImg = rhs.OriginalImg;
56:     this->ProcessedImg = rhs.ProcessedImg;
57:     this->TempImg = rhs.ProcessedImg;
58: }
59:
60: MorphologicalFilter::~MorphologicalFilter() {}
61:
62: MorphologicalFilter &MorphologicalFilter::operator=(MorphologicalFilter &rhs) {
63:     if (&rhs != this) {
64:         this->OriginalImg = rhs.OriginalImg;
65:         this->ProcessedImg = rhs.ProcessedImg;
66:         this->TempImg = rhs.TempImg;
67:     }
68:     return *this;
69: }
70:
71: void MorphologicalFilter::Open(const Mat &mask, bool chain) {
72:     Erosion(mask, chain);
73:     Dilation(mask, true);
74: }
75:
76: void MorphologicalFilter::Close(const Mat &mask, bool chain) {
77:     Dilation(mask, chain);
78:     Erosion(mask, true);
79: }
80:
81: void MorphologicalFilter::Dilation(const Mat &mask, bool chain) {
82:     Filter(mask, chain, 0, 1, 1);
83: }
```



```

84:
85: void MorphologicalFilter::Erosion(const Mat &mask, bool chain) {
86:     Filter(mask, chain, 1, 0, 0);
87: }
88:
89: void MorphologicalFilter::Filter(const Mat &mask, bool chain, uchar startVal,
90:                                 uchar newVal, uchar switchVal) {
91:     // Exception handling
92:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
93:     EMPTY_CHECK(OriginalImg);
94:     if (mask.cols % 2 == 0 || mask.cols < 3) {
95:         throw Exception::WrongKernelSizeException("Wrong Kernel size columns!");
96:     }
97:     if (mask.rows % 2 == 0 || mask.rows < 3) {
98:         throw Exception::WrongKernelSizeException("Wrong Kernel size rows!");
99:     }
100:
101:     uint32_t hKSizeCol = (mask.cols / 2);
102:     uint32_t hKSizeRow = (mask.rows / 2);
103:
104:     // make Pointers
105:     Mat workOrigImg(ProcessedImg.rows + mask.rows, ProcessedImg.cols + mask.cols,
106:                     CV_8UC1);
107:     workOrigImg.setTo(0);
108:     if (chain) {
109:         ProcessedImg.copyTo(workOrigImg(
110:             cv::Rect(hKSizeCol, hKSizeRow, ProcessedImg.cols, ProcessedImg.rows)));
111:         // workOrigImg(cv::Rect(hKSizeCol, hKSizeRow, ProcessedImg.cols,
112:         // ProcessedImg.rows)) = ProcessedImg.clone();
113:     } else {
114:         OriginalImg.copyTo(workOrigImg(
115:             cv::Rect(hKSizeCol, hKSizeRow, ProcessedImg.cols, ProcessedImg.rows)));
116:         // workOrigImg(cv::Rect(hKSizeCol, hKSizeRow, ProcessedImg.cols,
117:         // ProcessedImg.rows)) = OriginalImg.clone();
118:     }
119:     uchar *O = workOrigImg.data;
120:
121:     Mat workProcImg(ProcessedImg.rows + mask.rows, ProcessedImg.cols + mask.cols,
122:                     CV_8UC1);
123:     uchar *P = workProcImg.data;
124:
125:     // Init the relevant data
126:     //uint32_t nData = OriginalImg.cols * OriginalImg.rows;
127:     uint32_t nWData = workProcImg.cols * workProcImg.rows;
128:     uint32_t nWStart = (hKSizeRow * workProcImg.cols) + hKSizeRow;
129:     uint32_t nWEnd = nWData - hKSizeCol - hKSizeRow * workProcImg.cols - 1;
130:     uchar *nRow = GetNRow(nWData, hKSizeCol, workProcImg.cols, workProcImg.rows);
131:     int MaskPixel = 0, OPixel = 0;
132:
133:     workProcImg.setTo(0);
134:     if (startVal != 0) {
135:         workProcImg(cv::Rect(hKSizeCol, hKSizeRow, ProcessedImg.cols,
136:                               ProcessedImg.rows)).setTo(startVal);
137:     }
138:     SHOW_DEBUG_IMG(workOrigImg, uchar, 255, "workOrigImg Filter!", false);
139:     SHOW_DEBUG_IMG(mask, uchar, 255, "Filter mask", true);
140:
141:     for (uint32_t i = nWStart; i < nWEnd; i++) {
142:         // Checks if pixel isn't a border pixel and progresses to the new row
143:         if (nRow[i] == 1) {
144:             i += mask.cols;
145:         }
146:         for (int r = 0; r < mask.rows; r++) {
147:             for (int c = 0; c < mask.cols; c++) {
148:                 MaskPixel = c + r * mask.cols;
149:                 OPixel = i - hKSizeCol + c + (r - hKSizeRow) * workProcImg.cols;
150:                 if (mask.data[MaskPixel] == 1 && O[OPixel] == switchVal) {
151:                     P[i] = newVal;
152:                     c = mask.cols;
153:                     r = mask.rows;
154:                 }
155:             }
156:         }
157:     }
158:     delete[] nRow;
159:     SHOW_DEBUG_IMG(workProcImg, uchar, 255, "workProcImg Filter!", true);
160:     ProcessedImg = workProcImg(Rect(hKSizeCol, hKSizeRow, ProcessedImg.cols,
161:                                     ProcessedImg.rows)).clone();
162:     SHOW_DEBUG_IMG(ProcessedImg, uchar, 255, "Processed Image Filter!", true);
163: }
164: }

```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  4:  * This software is proprietary and confidential
5:  5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  6:  */
7:
8:  #pragma once
9:  9:  /*! Current class version*/
10: #define IMAGEPROCESSING_VERSION 1
11:
12: 12: /*! MACRO which sets the original pointer to the original image or a clone of
13: 13: * the earlier processed image */
14: #define CHAIN_PROCESS(chain, O, type)
15:     if (chain) {
16:         TempImg = ProcessedImg.clone();
17:         O = (type *)TempImg.data;
18:     } else {
19:         O = (type *)OriginalImg.data;
20:     }
21: 21: /*! MACRO which throws an EmptyImageException if the matrix is empty*/
22: #define EMPTY_CHECK(img)
23:     if (img.empty()) {
24:         throw Exception::EmptyImageException();
25:     }
26:
27: #include <opencv2/core.hpp>
28: #include <opencv2/highgui.hpp>
29:
30: #include <stdint.h>
31: #include <cmath>
32: #include <vector>
33: #include <string>
34:
35: #include <boost/signals2.hpp>
36: #include <boost/bind.hpp>
37:
38: #include "EmptyImageException.h"
39: #include "WrongKernelSizeException.h"
40: #include "ChannelMismatchException.h"
41: #include "PixelValueOutOfBoundsException.h"
42: #include "VisionDebug.h"
43:
44: using namespace cv;
45:
46: namespace Vision {
47: class ImageProcessing {
48: public:
49:     typedef boost::signals2::signal<void(float, std::string)> Progress_t;
50:     boost::signals2::connection
51:     connect_Progress(const Progress_t::slot_type &subscriber);
52:
53: protected:
54:     uchar *GetNRow(int nData, int hKsize, int nCols, uint32_t totalRows);
55:     Mat TempImg;
56:
57:     Progress_t prog_sig;
58:
59: public:
60:     ImageProcessing();
61:     ~ImageProcessing();
62:     Mat OriginalImg;
63:     Mat ProcessedImg;
64:
65:     double currentProg = 0.;
66:     double ProgStep = 0.;
67:
68:     static std::vector<Mat> extractChannel(const Mat &src);
69:
70:     70:     /*! Copy a matrix to a new matrix with a LUT mask
71:     71:     \param src the source image
72:     72:     \param *LUT type T with a LUT to filter out unwanted pixel values
73:     73:     \param cvType an in where you can pas CV_UC8C1 etc.
74:     74:     \return The new matrix
75:     75:     */
76:     template <typename T1, typename T2>
77:     static Mat CopyMat(const Mat &src, T1 *LUT, int cvType) {
78:         Mat dst(src.size(), cvType);
79:         uint32_t nData = src.rows * src.cols * dst.step[1];
80:         if (cvType == 0 || cvType == 8 || cvType == 16 || cvType == 24) {
81:             for (uint32_t i = 0; i < nData; i += dst.step[1]) {
82:                 dst.data[i] =
83:                     static_cast<uint8_t>(LUT[(T2 *)](src.data + (i * src.step[1]))));

```

```

84:     }
85: } else if (cvType == 1 || cvType == 9 || cvType == 17 || cvType == 25) {
86:     for (uint32_t i = 0; i < nData; i += src.step[1]) {
87:         dst.data[i] =
88:             static_cast<int8_t>(LUT[*](T2 *)(src.data + (i * src.step[1]))));
89:     }
90: } else if (cvType == 2 || cvType == 10 || cvType == 18 || cvType == 26) {
91:     for (uint32_t i = 0; i < nData; i += src.step[1]) {
92:         dst.data[i] =
93:             static_cast<uint16_t>(LUT[*](T2 *)(src.data + (i * src.step[1]))));
94:     }
95: } else if (cvType == 3 || cvType == 11 || cvType == 19 || cvType == 27) {
96:     for (uint32_t i = 0; i < nData; i += src.step[1]) {
97:         dst.data[i] =
98:             static_cast<int16_t>(LUT[*](T2 *)(src.data + (i * src.step[1]))));
99:     }
100: } else if (cvType == 4 || cvType == 12 || cvType == 20 || cvType == 28) {
101:     for (uint32_t i = 0; i < nData; i += src.step[1]) {
102:         dst.data[i] =
103:             static_cast<int32_t>(LUT[*](T2 *)(src.data + (i * src.step[1]))));
104:     }
105: }
106: return dst;
107: }
108:
109: /* Copy a matrix to a new matrix with a mask
110: \param src the source image
111: \param *LUT type T with a LUT to filter out unwanted pixel values
112: \param cvType an in where you can pass CV_8C1 etc.
113: \return The new matrix
114: */
115: template <typename T1>
116: static Mat CopyMat(const Mat &src, const Mat &mask, int cvType) {
117:     if (src.size != mask.size) {
118:         throw Exception::WrongKernelSizeException(
119:             "Mask not the same size as src Exception!");
120:     }
121:     if (mask.channels() != 1) {
122:         throw Exception::WrongKernelSizeException(
123:             "Mask has more then 1 channel Exception!");
124:     }
125:     Mat dst(src.size(), cvType);
126:
127:     vector<Mat> exSrc = Vision::ImageProcessing::extractChannel(src);
128:     vector<Mat> exDst;
129:
130:     int cvBaseType = cvType % 8;
131:     for_each(exSrc.begin(), exSrc.end(), [&](const Mat &sItem) {
132:         Mat dItem(src.size(), cvBaseType);
133:         std::transform(sItem.begin<T1>(), sItem.end<T1>(), mask.begin<T1>(),
134:             dItem.begin<T1>(),
135:             [](const T1 &s, const T1 &m) -> T1 { return s * m; });
136:         exDst.push_back(dItem);
137:     });
138:
139:     merge(exDst, dst);
140:
141:     return dst;
142: }
143:
144: template <typename T1>
145: static void ShowDebugImg(cv::Mat img, T1 maxVal, std::string windowName,
146:     bool scale = true) {
147:     if (img.rows > 0 && img.cols > 0) {
148:         cv::Mat tempImg(img.size(), img.type());
149:         if (scale == true) {
150:             std::vector<cv::Mat> exSrc = extractChannel(img);
151:             std::vector<cv::Mat> exDst;
152:             int cvBaseType = img.type() % 8;
153:             T1 MatMin = std::numeric_limits<T1>::max();
154:             T1 MatMax = std::numeric_limits<T1>::min();
155:
156:             // Find the global max and min
157:             for_each(exSrc.begin(), exSrc.end(), [&](const Mat &sItem) {
158:                 std::for_each(sItem.begin<T1>(), sItem.end<T1>(), [&](const T1 &s) {
159:                     if (s > MatMax) {
160:                         MatMax = s;
161:                     } else if (s < MatMin) {
162:                         MatMin = s;
163:                     }
164:                 });
165:             });
166:

```

```

167:         int Range = MatMax - MatMin;
168:         if (Range < 1)
169:             Range = maxVal;
170:
171:         // Convert the values
172:         for_each(exSrc.begin(), exSrc.end(), [&](const cv::Mat &sItem) {
173:             Mat dItem(img.size(), cv_BaseType);
174:             std::transform(sItem.begin<T1>(), sItem.end<T1>(), dItem.begin<T1>(),
175:                 [&](const T1 &s) -> T1 {
176:                     return (T1)round(((s - MatMin) * maxVal) / Range);
177:                 });
178:             exDst.push_back(dItem);
179:         });
180:
181:         merge(exDst, tempImg);
182:     } else {
183:         tempImg = img;
184:     }
185:     cv::namedWindow(windowName, cv::WINDOW_NORMAL);
186:     cv::imshow(windowName, tempImg);
187:     cv::waitKey(0);
188:     cv::destroyWindow(windowName);
189: };
190: };
191: };
192: }

```

```
1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  4:  * This software is proprietary and confidential
5:  5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  6:  */
7:
8:  #pragma once
9:
10: #include <vector>
11: #include <queue>
12: #include <string>
13: #include <stdint.h>
14: #include <iostream>
15: #include <algorithm>
16: #include <utility>
17:
18: #include <boost/range/adaptor/reversed.hpp>
19:
20: #include "opencv2/imgproc/imgproc.hpp"
21:
22: #include "ImageProcessing.h"
23: #include "MorphologicalFilter.h"
24: #include "../SoilMath/SoilMath.h"
25:
26: namespace Vision {
27: class Segment : public ImageProcessing {
28: public:
29:     /*! Coordinates for the region of interest*/
30:     typedef struct Rect {
31:         uint16_t leftX; /*!< Left X coordinate*/
32:         uint16_t leftY; /*!< Left Y coordinate*/
33:         uint16_t rightX; /*!< Right X coordinate*/
34:         uint16_t rightY; /*!< Right Y coordinate*/
35:         Rect(uint16_t lx, uint16_t ly, uint16_t rx, uint16_t ry)
36:             : leftX(lx), leftY(ly), rightX(rx), rightY(ry){};
37:     } Rect_t;
38:
39:     typedef std::vector<Vision::Segment::Rect_t> RectList_t;
40:
41:     /*! Individual blob*/
42:     typedef struct Blob {
43:         uint16_t Label; /*!< ID of the blob*/
44:         cv::Mat Img; /*!< BW image of the blob all the pixel belonging to the blob
45:             are set to 1 others are 0*/
46:         cv::Rect ROI; /*!< Coordinates for the blob in the original picture as a
47:             cv::Rect*/
48:         uint32_t Area; /*!< Calculated stats of the blob*/
49:         Blob(uint16_t label, uint32_t area) : Label(label), Area(area){};
50:     } Blob_t;
51:
52:     typedef std::vector<Blob_t> BlobList_t;
53:     BlobList_t BlobList; /*!< vector with all the individual blobs*/
54:
55:     /*! Enumerator to indicate what kind of object to extract */
56:     enum TypeOfObjects {
57:         Bright, /*!< Enum value Bright object */
58:         Dark /*!< Enum value Dark object. */
59:     };
60:
61:     /*! Enumerator to indicate how the pixel correlate between each other in a
62:     * blob*/
63:     enum Connected {
64:         Four =
65:             2, /*!< Enum Four connected, relation between Center, North, East, South
66:             and West*/
67:         Eight =
68:             4 /*!< Enum Eight connected, relation between Center, North, NorthEast,
69:             East, SouthEast, South, SouthWest, West and NorthWest */
70:     };
71:
72:     /*!< Enumerator which indicate which Segmentation technique should be used */
73:     enum SegmentationType {
74:         Normal, /*!< Segmentation looking at the intensity of an individual pixel */
75:         LabNeuralNet, /*!< Segmentation looking at the chromatic a* and b* of the
76:             processed pixel and it's surrounding pixels, feeding it in
77:             an Neural Net */
78:         GraphMinCut /*!< Segmentation using a graph function and the minimum cut */
79:     };
80:
81:     cv::Mat LabelledImg; /*!< Image with each individual blob labeled with a
82:             individual number */
83:     uint16_t MaxLabel = 0; /*!< Maximum labels found in the labelled image*/
```

```
84:  uint16_t noOfFilteredBlobs =
85:      0; /*< Total numbers of blobs that where filtered beacuse the where
86:          smaller than the minBlobArea*/
87:
88:  ucharStat_t OriginalImgStats; /*< Statistical data from the original image*/
89:  uint8_t ThresholdLevel = 0; /*< Current calculated threshold level*/
90:
91:  float sigma = 2;
92:  uint32_t thresholdOffset = 4;
93:
94:  Segment();
95:  Segment(const Mat &src);
96:  Segment(const Segment &rhs);
97:
98:  ~Segment();
99:
100: Segment &operator=(Segment &rhs);
101:
102: void LoadOriginalImg(const Mat &src);
103:
104: void ConvertToBW(TypeOfObjects Typeobjects);
105: void ConvertToBW(const Mat &src, Mat &dst, TypeOfObjects Typeobjects);
106:
107: void GetEdges(bool chain = false, Connected conn = Eight);
108: void GetEdges(const Mat &src, Mat &dst, bool chain = false,
109:              Connected conn = Eight);
110:
111: void GetEdgesEroding(bool chain = false);
112:
113: void GetBlobList(bool chain = false, Connected conn = Eight);
114:
115: void Threshold(uchar t, TypeOfObjects Typeobjects);
116:
117: void LabelBlobs(bool chain = false, uint16_t minBlobArea = 25,
118:                Connected conn = Eight);
119:
120: void RemoveBorderBlobs(uint32_t border = 1, bool chain = false);
121:
122: void FillHoles(bool chain = false);
123:
124: private:
125:  uint8_t GetThresholdLevel(TypeOfObjects TypeObject);
126:  void SetBorder(uchar *P, uchar setValue);
127:  void FloodFill(uchar *O, uchar *P, uint16_t x, uint16_t y, uchar fillValue,
128:               uchar OldValue);
129:  void MakeConsecutive(uint16_t *valueArr, uint32_t noElem, uint16_t &maxlabel);
130:  void MakeConsecutive(uint16_t *valueArr, uint16_t *keyArr, uint16_t noElem,
131:                       uint16_t &maxlabel);
132:  void SortAdjacencyList(std::vector<std::vector<uint16_t>> &adj);
133:  void ConnectedBlobs(uchar *O, uint16_t *P,
134:                     std::vector<std::vector<uint16_t>> &adj, uint32_t nCols,
135:                     uint32_t nRows, Connected conn);
136:  void InvertAdjacencyList(std::vector<std::vector<uint16_t>> &adj,
137:                           std::vector<std::vector<uint16_t>> &adjInv);
138: };
139: }
```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class Enhance
9:  class which enhances a greyscale cv::Mat image
10: */
11: #include "Enhance.h"
12:
13: namespace Vision {
14:  /*! Constructor*/
15:  Enhance::Enhance() {}
16:
17:  /*! Constructor
18:  \param src cv::Mat source image
19:  */
20:  Enhance::Enhance(const Mat &src) {
21:      OriginalImg = src;
22:      ProcessedImg.create(OriginalImg.size(), CV_8UC1);
23:  }
24:
25:  Enhance::Enhance(const Enhance &rhs) {
26:      this->OriginalImg = rhs.OriginalImg;
27:      this->ProcessedImg = rhs.OriginalImg;
28:      this->TempImg = rhs.TempImg;
29:  }
30:
31:  /*! Constructor
32:  \param src cv::Mat source image
33:  \param dst cv::Mat destination image
34:  \param kernelsize an uchar which represent the kernelsize should be an uneven
35:  number higher than two
36:  \param factor float which indicates the amount the effect should take place
37:  standard value is 1.0 only used in the adaptive contrast stretch enhancement
38:  \param operation enumerator EnhanceOperation which enhancement should be
39:  performed
40:  */
41:  Enhance::Enhance(const Mat &src, Mat &dst, uchar kernelsize, float factor,
42:      EnhanceOperation operation) {
43:      OriginalImg = src;
44:      ProcessedImg.create(OriginalImg.size(), CV_8UC1);
45:      switch (operation) {
46:      case Vision::Enhance::_AdaptiveContrastStretch:
47:          AdaptiveContrastStretch(kernelsize, factor);
48:          break;
49:      case Vision::Enhance::_Blur:
50:          Blur(kernelsize);
51:          break;
52:      case Vision::Enhance::_HistogramEqualization:
53:          HistogramEqualization();
54:          break;
55:      }
56:      dst = ProcessedImg;
57:  }
58:
59:  /*! Dec-constructor*/
60:  Enhance::~Enhance() {}
61:
62:  Enhance &Enhance::operator=(Enhance rhs) {
63:      if (&rhs != this) {
64:          this->OriginalImg = rhs.OriginalImg;
65:          this->ProcessedImg = rhs.ProcessedImg;
66:          this->TempImg = rhs.ProcessedImg;
67:      }
68:      return *this;
69:  }
70:
71:  /*! Calculate the standard deviation of the neighboring pixels
72:  \param O uchar pointer to the current pixel of the original image
73:  \param i current counter
74:  \param hKsize half the kernelsize
75:  \param nCols total number of columns
76:  \param noNeighboursPix total number of neighboring pixels
77:  \param mean mean value of the neighboring pixels
78:  \return standard deviation
79:  */
80:  float Enhance::CalculateStdOfNeighboringPixels(uchar *O, int i, int hKsize,
81:      int nCols, int noNeighboursPix,
82:      float mean) {
83:      uint32_t sum_dev = 0.0;

```

```
84: float Std = 0.0;
85: sum_dev = 0.0;
86: Std = 0.0;
87: for (int j = -hKsize; j < hKsize; j++) {
88:     for (int k = -hKsize; k < hKsize; k++) {
89:         // sum_dev += pow((O[i + j * nCols + k] - mean), 2);
90:         sum_dev += SoilMath::quick_pow2((O[i + j * nCols + k] - mean));
91:     }
92: }
93: // Std = sqrt(sum_dev / noNeighboursPix);
94: Std = SoilMath::fastPow((static_cast<double>(sum_dev) / noNeighboursPix), 2);
95: return Std;
96: }
97:
98: /*! Calculate the sum of the neighboring pixels
99: \param O uchar pointer to the current pixel of the original image
100: \param i current counter
101: \param hKsize half the kernel size
102: \param nCols total number of columns
103: \param sum Total sum of the neighboring pixels
104: */
105: void Enhance::CalculateSumOfNeighboringPixels(uchar *O, int i, int hKsize,
106:                                              int nCols, uint32_t &sum) {
107:     for (int j = -hKsize; j < hKsize; j++) {
108:         for (int k = -hKsize; k < hKsize; k++) {
109:             sum += O[i + j * nCols + k];
110:         }
111:     }
112: }
113:
114: /*! Homebrew AdaptiveContrastStretch function which calculate the mean and
115: standard deviation from the neighboring pixels if the current pixel is higher
116: then the mean the value is incremented with an given factor multiplied with the
117: standard deviation, and decreased if it's lower then the mean.
118: \param src cv::Mat source image
119: \param dst cv::Mat destination image
120: \param kernel size an uchar which represent the kernel size should be an uneven
121: number higher than two
122: \param factor float which indicates the amount the effect should take place
123: standard value is 1.0 only used in the adaptive contrast stretch enhancement
124: */
125: void Enhance::AdaptiveContrastStretch(const Mat &src, Mat &dst,
126:                                       uchar kernel size, float factor) {
127:     OriginalImg = src;
128:     ProcessedImg.create(OriginalImg.size(), CV_8UC1);
129:     AdaptiveContrastStretch(kernel size, factor);
130:     dst = ProcessedImg;
131: }
132:
133: /*! Homebrew AdaptiveContrastStretch function which calculate the mean and
134: standard deviation from the neighboring pixels if the current pixel is higher
135: then the mean the value is incremented with an given factor multiplied with the
136: standard deviation, and decreased if it's lower then the mean.
137: \param kernel size an uchar which represent the kernel size should be an uneven
138: number higher than two
139: \param factor float which indicates the amount the effect should take place
140: standard value is 1.0 only used in the adaptive contrast stretch enhancement
141: \param chain use the results from the previous operation default value = false;
142: */
143: void Enhance::AdaptiveContrastStretch(uchar kernel size, float factor,
144:                                       bool chain) {
145:     // Exception handling
146:     EMPTY_CHECK(OriginalImg);
147:     if (kernel size < 3 || (kernel size % 2) == 0) {
148:         throw Exception::WrongKernelSizeException();
149:     }
150:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
151:
152:     // Make the pointers to the Data
153:     uchar *O;
154:     CHAIN_PROCESS(chain, O, uchar);
155:     uchar *P = ProcessedImg.data;
156:
157:     int i = 0;
158:     int hKsize = kernel size / 2;
159:     int nCols = OriginalImg.cols;
160:     int pStart = (hKsize * nCols) + hKsize + 1;
161:
162:     int nData = OriginalImg.rows * OriginalImg.cols;
163:     int pEnd = nData - pStart;
164:     uint32_t noNeighboursPix = kernel size * kernel size;
165:     uint32_t sum;
166:     float mean = 0.0;
```



```
167:
168:   uchar *nRow = GetNRow(nData, hKsize, nCols, OriginalImg.rows);
169:
170:   i = pStart;
171:   while (i++ < pEnd) {
172:       // Checks if pixel isn't a border pixel and progresses to the new row
173:       if (nRow[i] == 1) {
174:           i += kernelsize;
175:       }
176:
177:       // Fill the neighboring pixel array
178:       sum = 0;
179:       mean = 0;
180:
181:       // Calculate the statistics
182:       CalculateSumOfNeighboringPixels(O, i, hKsize, nCols, sum);
183:       mean = (float)(sum / noNeighboursPix);
184:       float Std = CalculateStdOfNeighboringPixels(O, i, hKsize, nCols,
185:                                                   noNeighboursPix, mean);
186:
187:       // Stretch
188:
189:       if (O[i] > mean) {
190:           // int addValue = O[i] + (int)(round(factor * Std));
191:           int addValue = O[i] + static_cast<int>(round(factor * Std));
192:           if (addValue < 255) {
193:               P[i] = addValue;
194:           } else {
195:               P[i] = 255;
196:           }
197:       } else if (O[i] < mean) {
198:           // int subValue = O[i] - (int)(round(factor * Std));
199:           int subValue = O[i] - static_cast<int>(round(factor * Std));
200:           if (subValue > 0) {
201:               P[i] = subValue;
202:           } else {
203:               P[i] = 0;
204:           }
205:       } else {
206:           P[i] = O[i];
207:       }
208:   }
209:
210:   // Stretch the image with an normal histogram equalization
211:   HistogramEqualization(true);
212:
213:   delete[] nRow;
214: }
215:
216: /*! Blurs the image with a NxN kernel
217: \param src cv::Mat source image
218: \param dst cv::Mat destination image
219: \param kernelsize an uchar which represent the kernelsize should be an uneven
220: number higher than two
221: */
222: void Enhance::Blur(const Mat &src, Mat &dst, uchar kernelsize) {
223:     OriginalImg = src;
224:     ProcessedImg.create(OriginalImg.size(), CV_8UC1);
225:     Blur(kernelsize);
226:     dst = ProcessedImg;
227: }
228:
229: /*! Blurs the image with a NxN kernel
230: \param kernelsize an uchar which represent the kernelsize should be an uneven
231: number higher than two
232: \param chain use the results from the previous operation default value = false;
233: */
234: void Enhance::Blur(uchar kernelsize, bool chain) {
235:     // Exception handling
236:     EMPTY_CHECK(OriginalImg);
237:     if (kernelsize < 3 || (kernelsize % 2) == 0) {
238:         throw Exception::WrongKernelSizeException();
239:     }
240:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
241:
242:     // Make the pointers to the Data
243:     uchar *O;
244:     CHAIN_PROCESS(chain, O, uchar);
245:     uchar *P = ProcessedImg.data;
246:
247:     int nData = OriginalImg.rows * OriginalImg.cols;
248:     int hKsize = kernelsize / 2;
249:     int nCols = OriginalImg.cols;
```

```

250:  int pStart = (hKsize * nCols) + hKsize + 1;
251:  int pEnd = nData - pStart;
252:  int noNeighboursPix = kernelsize * kernelsize;
253:  uint32_t sum;
254:
255:  uint32_t i;
256:  uchar *nRow = GetNRow(nData, hKsize, nCols, OriginalImg.rows);
257:  i = pStart;
258:  while (i++ < pEnd) {
259:      // Checks if pixel isn't a border pixel and progresses to the new row
260:      if (nRow[i] == 1) {
261:          i += kernelsize;
262:      }
263:
264:      // Calculate the sum of the kernel
265:      sum = 0;
266:      CalculateSumOfNeighboringPixels(O, i, hKsize, nCols, sum);
267:
268:      P[i] = (uchar)(round(sum / noNeighboursPix));
269:  }
270:
271:  delete[] nRow;
272: }
273:
274: /* Stretches the image using a histogram
275: \param chain use the results from the previous operation default value = false;
276: */
277: void Enhance::HistogramEqualization(bool chain) {
278:     // Exception handling
279:     EMPTY_CHECK(OriginalImg);
280:     CV_Assert(OriginalImg.depth() != sizeof(uchar));
281:
282:     // Make the pointers to the Data
283:     uchar *O;
284:     CHAIN_PROCESS(chain, O, uchar);
285:     uchar *P = ProcessedImg.data;
286:
287:     // Calculate the statics of the whole image
288:     ucharStat_t imgStats(O, OriginalImg.rows, OriginalImg.cols);
289:     float sFact;
290:     if (imgStats.min != imgStats.max) {
291:         sFact = 255.0f / (imgStats.max - imgStats.min);
292:     } else {
293:         sFact = 1.0f;
294:     }
295:
296:     uint32_t i = 256;
297:     uchar LUT_changeValue[256];
298:     while (i-- > 0) {
299:         LUT_changeValue[i] = (uchar)(((float)(i)*sFact) + 0.5f);
300:     }
301:
302:     O = OriginalImg.data;
303:
304:     i = OriginalImg.cols * OriginalImg.rows + 1;
305:     while (i-- > 0) {
306:         *P++ = LUT_changeValue[*O++ - imgStats.min];
307:     }
308: }
309: }

```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  4:  * This software is proprietary and confidential
5:  5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  6:  */
7:
8:  #pragma once
9:  #define ENHANCE_VERSION 1
10:
11:  #include "ImageProcessing.h"
12:  #include "../SoilMath/SoilMath.h"
13:
14:  using namespace std;
15:  using namespace SoilMath;
16:
17:  namespace Vision {
18:  class Enhance : public ImageProcessing {
19:  private:
20:      void CalculateSumOfNeighboringPixels(uchar *O, int i, int hKsize, int nCols,
21:                                           uint32_t &sum);
22:      float CalculateStdOfNeighboringPixels(uchar *O, int i, int hKsize, int nCols,
23:                                           int noNeighboursPix, float mean);
24:
25:  public:
26:      /*! Enumerator indicating the requested enhancement operation*/
27:      enum EnhanceOperation {
28:          _AdaptiveContrastStretch, /*!< custom adaptive contrast stretch operation*/
29:          _Blur, /*!< Blur operation*/
30:          _HistogramEqualization /*!< Histogram equalization*/
31:      };
32:
33:      Enhance();
34:      Enhance(const Mat &src);
35:      Enhance(const Mat &src, Mat &dst, uint8_t kernelSize = 9, float factor = 1.0,
36:              EnhanceOperation operation = _Blur);
37:      Enhance(const Enhance &rhs);
38:
39:      ~Enhance();
40:
41:      Enhance &operator=(Enhance rhs);
42:
43:      void AdaptiveContrastStretch(uint8_t kernelSize, float factor,
44:                                   bool chain = false);
45:      void AdaptiveContrastStretch(const Mat &src, Mat &dst, uint8_t kernelSize,
46:                                   float factor);
47:
48:      void Blur(uint8_t kernelSize, bool chain = false);
49:      void Blur(const Mat &src, Mat &dst, uint8_t kernelSize);
50:
51:      void HistogramEqualization(bool chain = false);
52:      void HistogramEqualization(const Mat &src, Mat &dst);
53:  };
54: }

```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  /*! \class ImageProcessing
9:  \brief Core class of all the image classes
10: Core class of all the image classes with a few commonly shared functions and
11: variables
12: */
13: #include "ImageProcessing.h"
14:
15: namespace Vision {
16:  /*! Constructor of the core class*/
17:  ImageProcessing::ImageProcessing() {}
18:
19:  /*! De-constructor of the core class*/
20:  ImageProcessing::~ImageProcessing() {}
21:
22:  /*! Create a LUT indicating which iteration variable i is the end of an row
23:  \param nData an int indicating total pixels
24:  \param hKsize int half the size of the kernel, if any. which acts as an offset
25:  from the border pixels
26:  \param nCols int number of columns in a row
27:  \return array of uchars where a zero is a middle column and a 1 indicates an end
28:  of an row minus the offset from half the kernel size
29:  */
30:  uchar *ImageProcessing::GetNRow(int nData, int hKsize, int nCols,
31:                                   uint32_t totalRows) {
32:      // Create LUT to determine when there is an new row
33:      uchar *nRow = new uchar[nData + 1]{};
34:      // int i = 0;
35:      int shift = nCols - hKsize - 1;
36:      for (uint32_t i = 0; i < totalRows; i++) {
37:          nRow[(i * nCols) + shift] = 1;
38:      }
39:      return nRow;
40:  }
41:
42:  std::vector<Mat> ImageProcessing::extractChannel(const Mat &src) {
43:      vector<Mat> chans;
44:      split(src, chans);
45:      return chans;
46:  }
47:
48:  boost::signals2::connection
49:  ImageProcessing::connect_Progress(const Progress_t::slot_type &subscriber) {
50:      return prog_sig.connect(subscriber);
51:  }
52: }

```

```

1:  /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8:  #pragma once
9:  #include "ImageProcessing.h"
10: #include "ConversionNotSupportedException.h"
11:
12: namespace Vision {
13:     class Conversion : public ImageProcessing {
14:     public:
15:         /*! Enumerator which indicates the colorspace used*/
16:         enum ColorSpace {
17:             CIE_lab,      /*!< CIE La*b* colorspace */
18:             CIE_XYZ,      /*!< CIE XYZ colorspace */
19:             RI,           /*!< Redness Index colorspace */
20:             RGB,          /*!< RGB colorspace */
21:             Intensity,    /*!< Grayscale colorspace */
22:             None,         /*!< none */
23:         };
24:         ColorSpace OriginalColorSpace; /*!< The original colorspace*/
25:         ColorSpace ProcessedColorSpace; /*!< The destination colorspace*/
26:
27:         Conversion();
28:         Conversion(const Mat &src);
29:         Conversion(const Conversion &rhs);
30:
31:         ~Conversion();
32:
33:         Conversion &operator=(Conversion rhs);
34:
35:         void Convert(ColorSpace convertFrom, ColorSpace convertTo,
36:                     bool chain = false);
37:         void Convert(const Mat &src, Mat &dst, ColorSpace convertFrom,
38:                     ColorSpace convertTo, bool chain = false);
39:
40:     private:
41:         /*!< Conversion matrix used in the conversion between RGB and CIE XYZ*/
42:         float XYZmat[3][3] = {{0.412453, 0.357580, 0.180423},
43:                               {0.212671, 0.715160, 0.072169},
44:                               {0.019334, 0.119194, 0.950227}};
45:
46:         float whitePoint[3] = {
47:             0.9504, 1.0000, 1.0889}; /*!< Natural whitepoint in XYZ colorspace D65
48:                                     according to Matlab */
49:         // float whitePoint[3] = { 0.9642, 1.0000, 0.8251 }; /*!< Natural whitepoint
50:         // in XYZ colorspace D50 according to Matlab */
51:
52:         void Lab2RI(float *O, float *P, int nData);
53:         void RGB2XYZ(uchar *O, float *P, int nData);
54:         void XYZ2Lab(float *O, float *P, int nData);
55:         void RGB2Intensity(uchar *O, uchar *P, int nData);
56:         inline float f_xyz2lab(float t);
57:     };
58: }

```