```cpp
 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 Conversion
 9: class which converts a cv::Mat image from one colorspace to the next colorspace
10: */
11: #include "Conversion.h"
12: namespace Vision {
13: /*! Constructor of the class    */
14: Conversion::Conversion() {
15:   OriginalColorSpace = None;
16:   ProcessedColorSpace = None;
17: }
18:
19: /*! Constructor of the class
20: \param src a cv::Mat object which is the source image
21: */
22: Conversion::Conversion(const Mat &src) {
23:   OriginalColorSpace = None;
24:   ProcessedColorSpace = None;
25:   OriginalImg = src;
26: }
27:
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: /*! De-constructor of the class*/
38: Conversion::~Conversion() {}
39:
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: /*! Convert the source image from one colorspace to a destination colorspace
53: - RGB 2 Intensity
54: - RGB 2 XYZ
55: - RGB 2 Lab
56: - RGB 2 Redness Index
57: - XYZ 2 Lab
58: - XYZ 2 Redness Index
59: - Lab 2 Redness Index
60: \param src a cv::Mat object which is the source image
61: \param dst a cv::Mat object which is the destination image
62: \param convertFrom the starting colorspace
63: \param convertTo the destination colorspace
64: \param chain use the results from the previous operation default value = false;
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: /*! Convert the source image from one colorspace to a destination colorspace
74: posibilities are:
75: - RGB 2 Intensity
76: - RGB 2 XYZ
77: - RGB 2 Lab
78: - RGB 2 Redness Index
79: - XYZ 2 Lab
80: - XYZ 2 Redness Index
81: - Lab 2 Redness Index
82: \param convertFrom the starting colorspace
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;
```

```cpp
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: }
```

```cpp
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: }
```

```cpp
 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 kernelsize 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: }
```

```cpp
 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: }
```

```cpp
 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: }
```

```cpp
  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);
```

```cpp
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 tiping 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;
```

```cpp
250:    CHAIN_PROCESS(chain, O, 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;
```

```cpp
333:    uint32_t nRows = OriginalImg.rows;
334:    uint32_t nData = nCols * nRows;
335:
336:    // Determine the size of the array for beginning and endrow and middle of a
337:    // row
338:    uint32_t noConn[3] = {static_cast<uint32_t>(conn),
339:                          (static_cast<uint32_t>(conn) / 2),
340:                          (static_cast<uint32_t>(conn) / 2) + 1};
341:    uint32_t lastConn[3] = {noConn[0] - 1, noConn[1] - 1, noConn[2] - 1};
342:
343:    uint16_t currentlbl = 0;
344:    vector<vector<uint16_t>> CL;
345:    vector<uint16_t> zeroVector;
346:    zeroVector.push_back(currentlbl);
347:    CL.push_back(zeroVector);
348:
349:    // Determine which borderpixels should be handled differently
350:    uchar *nRow = new uchar[nData]{};
351:    for (uint32_t i = nCols; i < nData; i += nCols) {
352:      nRow[i] = 1;
353:      nRow[i - 1] = 2;
354:    }
355:
356:    // Set the first pixel
357:    if (O[0] == 0) {
358:      P[0] = 0;
359:    } else if (O[0] == 1) {
360:      P[0] = 1;
361:    } else {
362:      throw Exception::PixelValueOutOfBoundException();
363:    }
364:
365:    // Walk through the toprow and determine if it's a new blob or it's connected
366:    // with previously determine blob
367:    for (uint32_t i = 1; i < nCols; i++) {
368:      if (O[i] == 0) {
369:        P[i] = 0;
370:      } else if (O[i] == 1) {
371:        // If West is zero assume this is a new blob
372:        if (P[i - 1] == 0) {
373:          P[i] = ++currentlbl;
374:          vector<uint16_t> cVector;
375:          cVector.push_back(currentlbl);
376:          CL.push_back(cVector);
377:        } else { // set as previous blob
378:          P[i] = P[i - 1];
379:        }
380:      } else { // Value of of bounds
381:        throw Exception::PixelValueOutOfBoundException();
382:      }
383:    }
384:
385:    // walk through each pixel and determine if it's a new blob or it's connected
386:    // with previously determine blob
387:    for (uint32_t i = OriginalImg.cols; i < nData; i++) {
388:      if (O[i] == 0) { // Original pixel = 0
389:        P[i] = 0;
390:      } else if (O[i] == 1) {
391:        // Get an array of Neighboring Pixels
392:        uint16_t *nPixels = new uint16_t[noConn[nRow[i]]];
393:        if (nRow[i] != 1) {
394:          nPixels[0] = P[i - 1];
395:        }
396:        uint32_t j = i - nCols - ((nRow[i] == 1) ? 0 : ((conn == Four) ? 0 : 1));
397:        for_each(nPixels + ((nRow[i] != 1) ? 1 : 0), nPixels + noConn[nRow[i]],
398:                 [&](uint16_t &N) { N = P[j++]; });
399:
400:        // Sort the neighbors for easier checking
401:        SoilMath::Sort::QuickSort<uint16_t>(nPixels, noConn[nRow[i]]);
402:
403:        // If all are zero assume this is a new blob
404:        if (nPixels[lastConn[nRow[i]]] == 0) {
405:          P[i] = ++currentlbl;
406:          vector<uint16_t> cVector;
407:          cVector.push_back(currentlbl);
408:          CL.push_back(cVector);
409:        } else {
410:          /* Sets the processed value to the smallest non-zero value and update
411:           * the connectedLabels */
412:          for (uint32_t j = 0; j < noConn[nRow[i]]; j++) {
413:            if (nPixels[j] > 0) {
414:              P[i] = nPixels[j];
415:              break;
```

```cpp
416:                }
417:             }
418:
419:          /* If previous blobs belong to different connected components set the
420:           * current processed value to the lowest value and remember that the
421:           * other values should be the lowest value*/
422:          if (P[i] != nPixels[lastConn[nRow[i]]]) {
423:             for (int j = lastConn[nRow[i]]; j >= 0; --j) {
424:                if (nPixels[j] <= P[i]) {
425:                   break;
426:                } else {
427:                   CL[nPixels[j]].push_back(P[i]);
428:                }
429:             }
430:          }
431:       }
432:       delete[] nPixels;
433:    } else {
434:       throw Exception::PixelValueOutOfBoundException();
435:    }
436: }
437: delete[] nRow;
438:
439: cv::Mat temp(LabelledImg.rows, LabelledImg.cols, CV_8UC1);
440: LabelledImg.convertTo(temp, CV_8UC1);
441: SHOW_DEBUG_IMG(temp, uchar, 255, "temp labelblobs truncat 255", false);
442:
443:
444: // Sort all the vectors and make unique,
445: uint32_t j = 0;
446: for_each(CL.begin(), CL.end(), [&](std::vector<uint16_t> &L) {
447:    std::sort(L.begin(), L.end());
448:    std::vector<uint16_t>::iterator it;
449:    it = std::unique(L.begin(), L.end());
450:    L.resize(std::distance(L.begin(), it));
451:    if (L.size() > 1) {
452:       for (std::vector<uint16_t>::iterator iter = L.begin(); iter != L.end();
453:             ++iter) {
454:          if (*iter == j) {
455:             L.erase(iter);
456:             break;
457:          }
458:       }
459:    }
460:    j++;
461: });
462:
463: // identify all the lowest values in the adjacent list
464: uint16_t *valueArr = new uint16_t[CL.size()];
465: uint16_t *keyArr = new uint16_t[CL.size()];
466:
467: for (int i = CL.size() - 1; i >= 0; --i) {
468:    std::vector<uint16_t *> route;
469:    uint16_t minVal = i;
470:    for (uint32_t j = 0; j < CL[i].size(); j++) {
471:
472:       // add the first node to the queue;
473:       route.push_back(&CL[i][j]);
474:
475:       // itterate till the last node
476:       bool lastNodeReached = false;
477:       while (!lastNodeReached) {
478:          uint32_t nodesVisited = route.size() - 1;
479:          if (*route[nodesVisited] < minVal) { minVal = *route[nodesVisited]; }
480:          route.push_back(&CL[*route[nodesVisited]][0]);
481:          if (route[nodesVisited] == route[nodesVisited + 1]) {
482:             route.pop_back();
483:             lastNodeReached = true;
484:          }
485:       }
486:
487:    }
488:    // Set all values to the lowest value
489:    for (uint32_t k = 0; k < route.size(); k++) {
490:       *route[k] = minVal;
491:    }
492:    valueArr[i] = minVal;
493:    keyArr[i] = i;
494: }
495:
496: // Make numbers consecutive
497: SoilMath::Sort::QuickSort<uint16_t>(valueArr, keyArr, CL.size());
498: uint16_t count = 0;
```

```cpp
499:    for (uint32_t i = 1; i < CL.size(); i++) {
500:      if (valueArr[i] != valueArr[i - 1]) {
501:        count++;
502:      }
503:      valueArr[i] = count;
504:    }
505:    SoilMath::Sort::QuickSort<uint16_t>(keyArr, valueArr, CL.size());
506:    delete[] keyArr;
507:    MaxLabel = count;
508:
509:    // Second loop through the pixels to give the values a final value
510:    for_each(P, P + nData, [&](uint16_t &V) { V = valueArr[V]; });
511:    delete[] valueArr;
512:    SHOW_DEBUG_IMG(LabelledImg, uint16_t, uint16_t(-1), "LabbeldImg", true);
513: }
514:
515: /*! Create a BW image with only edges from a BW image
516: \param src source image as a const cv::Mat
517: \param dst destination image as a cv::Mat
518: \param conn set the pixel connection eight or four
519: \param chain use the results from the previous operation default value = false;
520: */
521: void Segment::GetEdges(const Mat &src, Mat &dst, bool chain, Connected conn) {
522:    OriginalImg = src;
523:    GetEdges(chain, conn);
524:    dst = ProcessedImg;
525: }
526:
527: /*! Create a BW image with only edges from a BW image
528: \param conn set the pixel connection eight or four
529: \param chain use the results from the previous operation default value = false;
530: */
531: void Segment::GetEdges(bool chain, Connected conn) {
532:    // Exception handling
533:    CV_Assert(OriginalImg.depth() != sizeof(uchar));
534:    EMPTY_CHECK(OriginalImg);
535:
536:    // make Pointers
537:    uchar *O;
538:    CHAIN_PROCESS(chain, O, uchar);
539:    uchar *P = ProcessedImg.data;
540:
541:    uint32_t nCols = OriginalImg.cols;
542:    uint32_t nRows = OriginalImg.rows;
543:    uint32_t nData = nCols * nRows;
544:    uint32_t pEnd = nData + 1;
545:    uint32_t i = 0;
546:
547:    // Loop through the image and set each pixel which has a zero neighbor set it
548:    // to two.
549:    if (conn == Four) {
550:      // Loop through the picture
551:      while (i < pEnd) {
552:        // If current value = zero processed value = zero
553:        if (O[i] == 0) {
554:          P[i] = 0;
555:        }
556:        // If current value = 1 check North West, South and East and act
557:        // accordingly
558:        else if (O[i] == 1) {
559:          uchar *nPixels = new uchar[4];
560:          nPixels[0] = O[i - 1];
561:          nPixels[1] = O[i - nCols];
562:          nPixels[2] = O[i + 1];
563:          nPixels[3] = O[i + nCols];
564:
565:          // Sort the neighbors for easier checking
566:          SoilMath::Sort::QuickSort<uchar>(nPixels, 4);
567:          if (nPixels[0] == 0) {
568:            P[i] = 1;
569:          } else {
570:            P[i] = 0;
571:          }
572:        } else {
573:          throw Exception::PixelValueOutOfBoundException();
574:        }
575:        i++;
576:      }
577:    } else {
578:      // Loop through the picture
579:      while (i < pEnd) {
580:        // If current value = zero processed value = zero
581:        if (O[i] == 0) {
```

```cpp
582:           P[i] = 0;
583:         }
584:         // If current value = 1 check North West, South and East and act
585:         // accordingly
586:         else if (O[i] == 1) {
587:           uchar *nPixels = new uchar[8];
588:           nPixels[0] = O[i - 1];
589:           nPixels[1] = O[i - nCols];
590:           nPixels[2] = O[i - nCols - 1];
591:           nPixels[3] = O[i - nCols + 1];
592:           nPixels[4] = O[i + 1];
593:           nPixels[5] = O[i + nCols + 1];
594:           nPixels[6] = O[i + nCols];
595:           nPixels[7] = O[i + nCols - 1];
596:
597:           // Sort the neighbors for easier checking
598:           SoilMath::Sort::QuickSort<uchar>(nPixels, 8);
599:
600:           if (nPixels[0] == 0) {
601:             P[i] = 1;
602:           } else {
603:             P[i] = 0;
604:           }
605:         } else {
606:           throw Exception::PixelValueOutOfBoundException();
607:         }
608:         i++;
609:       }
610:     }
611: }
612:
613: void Segment::GetEdgesEroding(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:   uchar *P = ProcessedImg.data;
622:
623:   uint32_t nCols = OriginalImg.cols;
624:   uint32_t nRows = OriginalImg.rows;
625:   uint32_t nData = nCols * nRows;
626:
627:   // Setup the erosion
628:   MorphologicalFilter eroder;
629:   if (chain) {
630:     eroder.OriginalImg = TempImg;
631:   } else {
632:     eroder.OriginalImg = OriginalImg;
633:   }
634:   // Setup the processed image of the eroder
635:   eroder.ProcessedImg.create(OriginalImg.size(), CV_8UC1);
636:   eroder.ProcessedImg.setTo(0);
637:   // Setup the mask
638:   Mat mask(3, 3, CV_8UC1, 1);
639:   // Erode the image
640:   eroder.Erosion(mask, false);
641:
642:   // Loop through the image and set the not eroded pixels to zero
643:   for (uint32_t i = 0; i < nData; i++) {
644:     if (O[i] != eroder.ProcessedImg.data[i]) {
645:       P[i] = 1;
646:     } else {
647:       P[i] = 0;
648:     }
649:   }
650:
651:   // ProcessedImg = OriginalImg.clone() - eroder.ProcessedImg.clone();
652:
653:   SHOW_DEBUG_IMG(eroder.ProcessedImg, uchar, 255, "Eroded img Processed Image!",
654:                 true);
655:   SHOW_DEBUG_IMG(ProcessedImg, uchar, 255, "GetEdgesEroding Processed Image!",
656:                 true);
657: }
658:
659: /*! Create a BlobList subtracting each individual blob out of a Labelled image.
660: If the labelled image is empty build a new one with a BW image.
661: \param conn set the pixel connection eight or four
662: \param chain use the results from the previous operation default value = false;
663: */
664: void Segment::GetBlobList(bool chain, Connected conn) {
```

```cpp
665:    // Exception handling
666:    CV_Assert(OriginalImg.depth() != sizeof(uchar));
667:    EMPTY_CHECK(OriginalImg);
668:
669:    // If there isn't a labelledImg make one
670:    if (MaxLabel < 1) {
671:      LabelBlobs(chain, 5, conn);
672:    }
673:
674:    // Make an empty BlobList
675:    uint32_t nCols = OriginalImg.cols;
676:    uint32_t nRows = OriginalImg.rows;
677:    uint32_t nData = nCols * nRows;
678:    RectList_t rectList;
679:
680:    // Calculate Stats the statistics
681:    uint16Stat_t LabelStats((uint16_t *)LabelledImg.data, LabelledImg.cols,
682:                            LabelledImg.rows, MaxLabel + 1, 0, MaxLabel);
683:
684:    BlobList.reserve(LabelStats.EndBin);
685:    rectList.reserve(LabelStats.EndBin);
686:
687:    BlobList.push_back(Blob_t(0, 0));
688:    rectList.push_back(Rect_t(0, 0, 0, 0));
689:
690:    for (uint32_t i = 1; i < LabelStats.EndBin; i++) {
691:      BlobList.push_back(Blob_t(i, LabelStats.bins[i]));
692:      rectList.push_back(Rect_t(nCols, nRows, 0, 0));
693:    }
694:
695:    // make Pointers
696:    uint16_t *L = (uint16_t *)LabelledImg.data;
697:
698:    uint32_t currentX, currentY;
699:    // uint16_t leftX, leftY, rightX, rightY;
700:    // Loop through the labeled image and extract the Blobs
701:    for (uint32_t i = 0; i < nData; i++) {
702:      if (L[i] != 0) {
703:        /* Determine the current x and y value of the current blob and
704:        checks if it is min/max */
705:        currentY = i / nCols;
706:        currentX = i % nCols;
707:
708:        // Min value
709:        if (currentX < rectList[L[i]].leftX) {
710:          rectList[L[i]].leftX = currentX;
711:        }
712:        if (currentY < rectList[L[i]].leftY) {
713:          rectList[L[i]].leftY = currentY;
714:        }
715:
716:        // Max value
717:        if (currentX > rectList[L[i]].rightX) {
718:          rectList[L[i]].rightX = currentX;
719:        }
720:        if (currentY > rectList[L[i]].rightY) {
721:          rectList[L[i]].rightY = currentY;
722:        }
723:      }
724:    }
725:
726:    // Loop through the BlobList and finalize it
727:    uint8_t *LUT_filter = new uint8_t[MaxLabel + 1]{};
728:    for (uint32_t i = 1; i <= MaxLabel; i++) {
729:      LUT_filter[i] = 1;
730:      BlobList[i].ROI.y = rectList[i].leftY;
731:      BlobList[i].ROI.x = rectList[i].leftX;
732:      BlobList[i].ROI.height = rectList[i].rightY - rectList[i].leftY + 1;
733:      BlobList[i].ROI.width = rectList[i].rightX - rectList[i].leftX + 1;
734:      BlobList[i].Img = CopyMat<uint8_t, uint16_t>(
735:          LabelledImg(BlobList[i].ROI).clone(), LUT_filter, CV_8UC1);
736:      // SHOW_DEBUG_IMG(BlobList[i].Img, uchar, 255, "Blob", true);
737:      LUT_filter[i] = 0;
738:    }
739:    delete[] LUT_filter;
740:
741:    // Remove background blob
742:    BlobList.erase(BlobList.begin());
743:  }
744:
745:  void Segment::FillHoles(bool chain) {
746:    // Exception handling
747:    CV_Assert(OriginalImg.depth() != sizeof(uchar));
```

```cpp
748:    EMPTY_CHECK(OriginalImg);
749:
750:    // make Pointers
751:    uchar *O;
752:    CHAIN_PROCESS(chain, O, uchar);
753:    if (chain) {
754:      ProcessedImg = TempImg.clone();
755:    } else {
756:      ProcessedImg = OriginalImg.clone();
757:    }
758:
759:    uchar *P = ProcessedImg.data;
760:
761:    // Determine the starting point of the floodfill
762:    int itt = -1;
763:    while (P[++itt] != 0)
764:      ;
765:    uint16_t row = static_cast<uint16_t>(itt / OriginalImg.rows);
766:    uint16_t col = static_cast<uint16_t>(itt % OriginalImg.rows);
767:
768:    // Fill the outside
769:    // FloodFill(O, P, row, col, 2, 0);
770:
771:    try {
772:      cv::floodFill(ProcessedImg, cv::Point(col, row), cv::Scalar(2));
773:    } catch (cv::Exception &e) {
774:    }
775:
776:    // Set the unreached areas to 1 and the outside to 0;
777:    uchar LUT_newVal[3] = {1, 1, 0};
778:    uint32_t nData = OriginalImg.rows * OriginalImg.cols;
779:    uint32_t i = 0;
780:    while (i <= nData) {
781:      P[i] = LUT_newVal[P[i]];
782:      i++;
783:    }
784:  }
785:
786:  void Segment::FloodFill(uchar *O, uchar *P, uint16_t row, uint16_t col,
787:                          uchar fillValue, uchar OldValue) {
788:    if (row < 0 || row > OriginalImg.rows) {
789:      return;
790:    }
791:    if (col < 0 || col > OriginalImg.cols) {
792:      return;
793:    }
794:    if (P[col + row * OriginalImg.rows] == OldValue) {
795:      P[col + row * OriginalImg.rows] = fillValue;
796:      FloodFill(O, P, row + 1, col, fillValue, OldValue);
797:      FloodFill(O, P, row, col + 1, fillValue, OldValue);
798:      FloodFill(O, P, row - 1, col, fillValue, OldValue);
799:      FloodFill(O, P, row, col - 1, fillValue, OldValue);
800:    }
801:  }
802: }
```

```
 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"
```

```cpp
 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, T1, maxVal, windowName, scale)                    \
21:    Vision::ImageProcessing::ShowDebugImg<T1>(img, maxVal, windowName, scale)
22: #endif // !SHOW_DEBUG_IMG
23: #else
24: #ifndef SHOW_DEBUG_IMG
25: #define SHOW_DEBUG_IMG(img, T1, 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 PixelValueOutOfBoundException
 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 PixelValueOutOfBoundException : public std::exception {
22: public:
23:   PixelValueOutOfBoundException(string m = "Current pixel value out of bounds!")
24:       : msg(m){};
25:   ~PixelValueOutOfBoundException() _GLIBCXX_USE_NOEXCEPT{};
26:   const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
27:
28: private:
29:   string msg;
30: };
31: }
32: }
```

```cpp
 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 EmtpyImageException
 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 EmtpyImageException : public std::exception {
23: public:
24:   EmtpyImageException(string m = "Empty Image!") : msg(m){};
25:   ~EmtpyImageException() _GLIBCXX_USE_NOEXCEPT{};
26:   const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
27:
28: private:
29:   string msg;
30: };
31: }
32: }
```

```cpp
 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: }
```

```cpp
 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: }
```

```cpp
 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 Kernelsize columns!");
 96:   }
 97:   if (mask.rows % 2 == 0 || mask.rows < 3) {
 98:     throw Exception::WrongKernelSizeException("Wrong Kernelsize 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:  * 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: /*! Current class version*/
10: #define IMAGEPROCESSING_VERSION 1
11:
12: /*! MACRO which sets the original pointer to the original image or a clone of
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: /*! MACRO which trows an EmtpyImageException if the matrix is empty*/
22: #define EMPTY_CHECK(img)                                                    \
23:   if (img.empty()) {                                                        \
24:     throw Exception::EmtpyImageException();                                 \
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 "PixelValueOutOfBoundException.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:   /*! Copy a matrix to a new matrix with a LUT mask
71:   \param src the source image
72:   \param *LUT type T with a LUT to filter out unwanted pixel values
73:   \param cvType an in where you can pas CV_UC8C1 etc.
74:   \return The new matrix
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 pas CV_UC8C1 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(), cvBaseType);
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: }
```

```cpp
 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:
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 = 2, /*!< Enum Four connected, relation between Center, North, East, South
65:               and West*/
66:     Eight = 4 /*!< Enum Eight connected, relation between Center, North, NorthEast,
67:               East, SouthEast, South, SouthWest, West and NorthWest */
68:   };
69:
70:   /*!< Enumerator which indicate which Segmentation technique should be used */
71:   enum SegmentationType {
72:     Normal, /*!< Segmentation looking at the intensity of an individual pixel */
73:     LabNeuralNet, /*!< Segmentation looking at the chromatic a* and b* of the
74:                        processed pixel and it's surrounding pixels, feeding it in
75:                        an Neural Net */
76:     GraphMinCut /*!< Segmentation using a graph function and the minimum cut */
77:   };
78:
79:   cv::Mat LabelledImg;   /*!< Image with each individual blob labeled with a
80:                               individual number */
81:   uint16_t MaxLabel = 0; /*!< Maximum labels found in the labelled image*/
82:   uint16_t noOfFilteredBlobs =
83:       0; /*!< Total numbers of blobs that where filtered beacuse the where
```

```
 84:                  smaller than the minBlobArea*/
 85:
 86:    ucharStat_t OriginalImgStats; /*!< Statistical data from the original image*/
 87:    uint8_t ThresholdLevel = 0;   /*!< Current calculated threshold level*/
 88:
 89:    float sigma = 2;
 90:    uint32_t thresholdOffset = 4;
 91:
 92:    Segment();
 93:    Segment(const Mat &src);
 94:    Segment(const Segment &rhs);
 95:
 96:    ~Segment();
 97:
 98:    Segment &operator=(Segment &rhs);
 99:
100:    void LoadOriginalImg(const Mat &src);
101:
102:    void ConvertToBW(TypeOfObjects Typeobjects);
103:    void ConvertToBW(const Mat &src, Mat &dst, TypeOfObjects Typeobjects);
104:
105:    void GetEdges(bool chain = false, Connected conn = Eight);
106:    void GetEdges(const Mat &src, Mat &dst, bool chain = false,
107:               Connected conn = Eight);
108:
109:    void GetEdgesEroding(bool chain = false);
110:
111:    void GetBlobList(bool chain = false, Connected conn = Eight);
112:
113:    void Threshold(uchar t, TypeOfObjects Typeobjects);
114:
115:    void LabelBlobs(bool chain = false, uint16_t minBlobArea = 25,
116:                 Connected conn = Eight);
117:
118:    void RemoveBorderBlobs(uint32_t border = 1, bool chain = false);
119:
120:    void FillHoles(bool chain = false);
121:
122: private:
123:    uint8_t GetThresholdLevel(TypeOfObjects TypeObject);
124:    void SetBorder(uchar *P, uchar setValue);
125:    void FloodFill(uchar *O, uchar *P, uint16_t x, uint16_t y, uchar fillValue,
126:               uchar OldValue);
127:    void makeConsecutive(uint16_t LastLabelUsed, uint16_t *tempLUT,
128:                   uint16_t *LUT_newVal);
129: };
130: }
```

```
  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;
```

```cpp
 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 kernelsize
102: \param nCols total number of columns
103: \param sum Total sum of the neighboringpixels
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 kernelsize an uchar which represent the kernelsize 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 kernelsize, float factor) {
127:     OriginalImg = src;
128:     ProcessedImg.create(OriginalImg.size(), CV_8UC1);
129:     AdaptiveContrastStretch(kernelsize, 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 kernelsize an uchar which represent the kernelsize 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 kernelsize, float factor,
144:                                       bool chain) {
145:     // Exception handling
146:     EMPTY_CHECK(OriginalImg);
147:     if (kernelsize < 3 || (kernelsize % 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 = kernelsize / 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 = kernelsize * kernelsize;
165:     uint32_t sum;
166:     float mean = 0.0;
```

```cpp
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;
```

```cpp
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:  * 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 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: }
```

```cpp
 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: }
```

```cpp
  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: }
```