```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class Conversion
class which converts a cv::Mat image from one colorspace to the next colorspace
*/
#include "Conversion.h"
namespace Vision {
/*! Constructor of the class    */
Conversion::Conversion() {
  OriginalColorSpace = None;
  ProcessedColorSpace = None;
}

/*! Constructor of the class
\param src a cv::Mat object which is the source image
*/
Conversion::Conversion(const Mat &src) {
  OriginalColorSpace = None;
  ProcessedColorSpace = None;
  OriginalImg = src;
}

/*! Copy constructor*/
Conversion::Conversion(const Conversion &rhs) {
  this->OriginalColorSpace = rhs.OriginalColorSpace;
  this->OriginalImg = rhs.OriginalImg;
  this->ProcessedColorSpace = rhs.ProcessedColorSpace;
  this->ProcessedImg = rhs.ProcessedImg;
  this->TempImg = rhs.TempImg;
}

/*! De-constructor of the class*/
Conversion::~Conversion() {}

/*! Assignment operator*/
Conversion &Conversion::operator=(Conversion rhs) {
  if (&rhs != this) {
    this->OriginalColorSpace = rhs.OriginalColorSpace;
    this->OriginalImg = rhs.OriginalImg;
    this->ProcessedColorSpace = rhs.ProcessedColorSpace;
    this->ProcessedImg = rhs.ProcessedImg;
    this->TempImg = rhs.TempImg;
  }
  return *this;
}

/*! Convert the source image from one colorspace to a destination colorspace
- RGB 2 Intensity
- RGB 2 XYZ
- RGB 2 Lab
- RGB 2 Redness Index
- XYZ 2 Lab
- XYZ 2 Redness Index
- Lab 2 Redness Index
\param src a cv::Mat object which is the source image
\param dst a cv::Mat object which is the destination image
\param convertFrom the starting colorspace
\param convertTo the destination colorspace
\param chain use the results from the previous operation default value = false;
*/
void Conversion::Convert(const Mat &src, Mat &dst, ColorSpace convertFrom,
                         ColorSpace convertTo, bool chain) {
  OriginalImg = src;
  Convert(convertFrom, convertTo, chain);
  dst = ProcessedImg;
}

/*! Convert the source image from one colorspace to a destination colorspace
posibilities are:
- RGB 2 Intensity
- RGB 2 XYZ
- RGB 2 Lab
- RGB 2 Redness Index
- XYZ 2 Lab
- XYZ 2 Redness Index
- Lab 2 Redness Index
\param convertFrom the starting colorspace
\param convertTo the destination colorspace
```

```
\param chain use the results from the previous operation default value = false;
*/
void Conversion::Convert(ColorSpace convertFrom, ColorSpace convertTo,
                         bool chain) {
  OriginalColorSpace = convertFrom;
  ProcessedColorSpace = convertTo;

  // Exception handling
  EMPTY_CHECK(OriginalImg);
  currentProg = 0.;
  prog_sig(currentProg, "Converting colorspace");

  int nData = OriginalImg.rows * OriginalImg.cols;
  // uint32_t i, j;

  if (convertFrom == RGB && convertTo == Intensity) // RGB 2 Intensity
  {
    ProcessedImg.create(OriginalImg.size(), CV_8UC1);
    uchar *P = ProcessedImg.data;
    uchar *O;
    CHAIN_PROCESS(chain, O, uchar);

    prog_sig(currentProg, "RGB 2 Intensity conversion");
    RGB2Intensity(O, P, nData);
    currentProg += ProgStep;
    prog_sig(currentProg, "RGB 2 Intensity conversion Finished");
  } else if (convertFrom == RGB && convertTo == CIE_XYZ) // RGB 2 XYZ
  {
    ProcessedImg.create(OriginalImg.size(), CV_32FC3);
    float *P = (float *)ProcessedImg.data;
    uchar *O;
    CHAIN_PROCESS(chain, O, uchar);

    prog_sig(currentProg, "RGB 2 CIE XYZ conversion");
    RGB2XYZ(O, P, nData);
    currentProg += ProgStep;
    prog_sig(currentProg, "RGB 2 CIE XYZ conversion Finished");
  } else if (convertFrom == RGB && convertTo == CIE_lab) // RGB 2 Lab
  {
    ProcessedImg.create(OriginalImg.size(), CV_32FC3);
    float *P = (float *)ProcessedImg.data;
    uchar *O;
    CHAIN_PROCESS(chain, O, uchar);

    prog_sig(currentProg, "RGB 2 CIE XYZ conversion");
    RGB2XYZ(O, P, nData);
    currentProg += ProgStep;
    prog_sig(currentProg, "RGB 2 CIE XYZ conversion Finished");
    Convert(CIE_XYZ, CIE_lab, true);
  } else if (convertFrom == RGB && convertTo == RI) // RGB 2 RI
  {
    ProcessedImg.create(OriginalImg.size(), CV_32FC3);
    float *P = (float *)ProcessedImg.data;
    uchar *O;
    CHAIN_PROCESS(chain, O, uchar);

    prog_sig(currentProg, "RGB 2 CIE XYZ conversion");
    RGB2XYZ(O, P, nData);
    currentProg += ProgStep;
    prog_sig(currentProg, "RGB 2 CIE XYZ conversion Finished");
    Convert(CIE_XYZ, CIE_lab, true);
    Convert(CIE_lab, RI, true);
  } else if (convertFrom == CIE_XYZ && convertTo == CIE_lab) // XYZ 2 Lab
  {
    ProcessedImg.create(OriginalImg.size(), CV_32FC3);
    float *P = (float *)ProcessedImg.data;
    float *O;
    CHAIN_PROCESS(chain, O, float);

    prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion");
    XYZ2Lab(O, P, nData);
    currentProg += ProgStep;
    prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion Finished");
  } else if (convertFrom == CIE_XYZ && convertTo == RI) // XYZ 2 RI
  {
    ProcessedImg.create(OriginalImg.size(), CV_32FC3);
    float *P = (float *)ProcessedImg.data;
    float *O;
    CHAIN_PROCESS(chain, O, float);

    prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion");
    XYZ2Lab(O, P, nData);
    currentProg += ProgStep;
```

```
    prog_sig(currentProg, "CIE XYZ 2 CIE La*b* conversion Finished");
    Convert(CIE_lab, RI, true);
  } else if (convertFrom == CIE_lab && convertTo == RI) // Lab 2 RI
  {
    ProcessedImg.create(OriginalImg.size(), CV_32FC1);
    float *P = (float *)ProcessedImg.data;
    float *O;
    CHAIN_PROCESS(chain, O, float);

    prog_sig(currentProg, "CIE La*b* 2 Redness Index conversion");
    Lab2RI(O, P, nData * 3);
    currentProg += ProgStep;
    prog_sig(currentProg, "CIE La*b* 2 Redness Index conversion Finsihed");
  } else {
    throw Exception::ConversionNotSupportedException();
  }
}

/*! Conversion from RGB to Intensity
\param O a uchar pointer to the source image
\param P a uchar pointer to the destination image
\param nData an int indicating the total number of pixels
*/
void Conversion::RGB2Intensity(uchar *O, uchar *P, int nData) {
  uint32_t i;
  int j;
  i = 0;
  j = 0;
  while (j < nData) {
    P[j++] = (*(O + i + 2) * 0.2126 + *(O + i + 1) * 0.7152 +
              *(O + i) * 0.0722); // Grey value
    i += 3;
  }
}

/*! Conversion from RGB to CIE XYZ
\param O a uchar pointer to the source image
\param P a uchar pointer to the destination image
\param nData an int indicating the total number of pixels
*/
void Conversion::RGB2XYZ(uchar *O, float *P, int nData) {
  uint32_t endData = nData * OriginalImg.step.buf[1];
  float R, G, B;
  for (uint32_t i = 0; i < endData; i += OriginalImg.step.buf[1]) {
    R = static_cast<float>(*(O + i + 2) / 255.0f);
    B = static_cast<float>(*(O + i + 1) / 255.0f);
    G = static_cast<float>(*(O + i) / 255.0f);
    P[i] = (XYZmat[0][0] * R) + (XYZmat[0][1] * B) + (XYZmat[0][2] * G); // X
    P[i + 1] = (XYZmat[1][0] * R) + (XYZmat[1][1] * B) + (XYZmat[1][2] * G); // Y
    P[i + 2] = (XYZmat[2][0] * R) + (XYZmat[2][1] * B) + (XYZmat[2][2] * G); // Z
  }
}

/*! Conversion from CIE XYZ to CIE La*b*
\param O a uchar pointer to the source image
\param P a uchar pointer to the destination image
\param nData an int indicating the total number of pixels
*/
void Conversion::XYZ2Lab(float *O, float *P, int nData) {
  uint32_t endData = nData * 3;
  float yy0, xx0, zz0;
  for (size_t i = 0; i < endData; i += 3) {
    xx0 = *(O + i) / whitePoint[0];
    yy0 = *(O + i + 1) / whitePoint[1];
    zz0 = *(O + i + 2) / whitePoint[2];

    if (yy0 > 0.008856) {
      P[i] = (116 * pow(yy0, 0.333f)) - 16; // L
    } else {
      P[i] = 903.3 * yy0; // L
    }

    P[i + 1] = 500 * (f_xyz2lab(xx0) - f_xyz2lab(yy0));
    P[i + 2] = 200 * (f_xyz2lab(yy0) - f_xyz2lab(zz0));
  }
}

inline float Conversion::f_xyz2lab(float t) {
  if (t > 0.008856) {
    return pow(t, 0.3333333333f);
  }
  return 7.787 * t + 0.137931034482759f;
}
```

```

/*! Conversion from CIE La*b* to Redness Index
\param O a uchar pointer to the source image
\param P a uchar pointer to the destination image
\param nData an int indicating the total number of pixels
*/
void Conversion::Lab2RI(float *O, float *P, int nData) {
  uint32_t j = 0;
  float L, a, b;
  for (int i = 0; i < nData; i += 3) {
    L = *(O + i);
    a = *(O + i + 1);
    b = *(O + i + 2);
    P[j++] =
        (L * (pow((pow(a, 2.0f) + pow(b, 2.0f)), 0.5f) * (pow(10, 8.2f)))) /
        (b * pow(L, 6.0f));
  }
}
}
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class WrongKernelSizeException
Exception class which is thrown when a wrong kernelsize is requested
*/
#pragma once

#include <exception>
#include <string>

using namespace std;

namespace Vision {
namespace Exception {
class WrongKernelSizeException : public std::exception {
public:
  WrongKernelSizeException(string m = "Wrong kernel dimensions!") : msg(m){};
  ~WrongKernelSizeException() _GLIBCXX_USE_NOEXCEPT{};
  const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };

private:
  string msg;
};
}
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class ConversionNotSupportedException
Exception class which is thrown when an illegal conversion is requested.
*/
#pragma once

#include <exception>
#include <string>

using namespace std;

namespace Vision {
namespace Exception {
class ConversionNotSupportedException : public std::exception {
public:
  ConversionNotSupportedException(
      string m = "Requested conversion is not supported!")
      : msg(m){};
  ~ConversionNotSupportedException() _GLIBCXX_USE_NOEXCEPT{};
  const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };

private:
  string msg;
};
}
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class ChannelMismatchException
Exception class which is thrown when Extracted channel out of bounds exception
*/

#pragma once

#include <exception>
#include <string>

using namespace std;

namespace Vision {
namespace Exception {
class ChannelMismatchException : public std::exception {
public:
  ChannelMismatchException(
      string m = "Extracted channel out of bounds exception!")
      : msg(m){};
  ~ChannelMismatchException() _GLIBCXX_USE_NOEXCEPT{};
  const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };

private:
  string msg;
};
}
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class  Segment
\brief Segmentation algorithms
With this class, various segmentation routines can be applied to a greyscale or
black and white source image.
*/
#include "Segment.h"

namespace Vision {
//! Constructor of the Segmentation class
Segment::Segment() {}

//! Constructor of the Segmentation class
Segment::Segment(const Mat &src) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  LabelledImg.create(OriginalImg.size(), CV_16UC1);
}

Segment::Segment(const Segment &rhs) {
  this->BlobList = rhs.BlobList;
  this->LabelledImg = rhs.LabelledImg;
  this->MaxLabel = rhs.MaxLabel;
  this->noOfFilteredBlobs = rhs.noOfFilteredBlobs;
  this->OriginalImg = rhs.OriginalImg;
  this->OriginalImgStats = rhs.OriginalImgStats;
  this->ProcessedImg = rhs.ProcessedImg;
  this->TempImg = rhs.TempImg;
  this->ThresholdLevel = rhs.ThresholdLevel;
}

//! De-constructor
Segment::~Segment() {}

Segment &Segment::operator=(Segment &rhs) {
  if (&rhs != this) {
    this->BlobList = rhs.BlobList;
    this->LabelledImg = rhs.LabelledImg;
    this->MaxLabel = rhs.MaxLabel;
    this->noOfFilteredBlobs = rhs.noOfFilteredBlobs;
    this->OriginalImg = rhs.OriginalImg;
    this->OriginalImgStats = rhs.OriginalImgStats;
    this->ProcessedImg = rhs.ProcessedImg;
    this->TempImg = rhs.TempImg;
    this->ThresholdLevel = rhs.ThresholdLevel;
  }
  return *this;
}

void Segment::LoadOriginalImg(const Mat &src) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  LabelledImg.create(OriginalImg.size(), CV_16UC1);
}

/*! Determine the threshold level by iteration, between two distribution,
presumably back- and foreground. It works towards the average of the two
averages and finally sets the threshold with two time the standard deviation
from the mean of the set object
\param TypeObject is an enumerator indicating if the bright or the dark pixels
are the object and should be set to one
\return The threshold level as an uint8_t       */
uint8_t Segment::GetThresholdLevel(TypeOfObjects TypeObject) {
  // Exception handling
  EMPTY_CHECK(OriginalImg);
  CV_Assert(OriginalImg.depth() != sizeof(uchar));

  // Calculate the statistics of the whole picture
  ucharStat_t OriginalImgStats(OriginalImg.data, OriginalImg.rows,
                               OriginalImg.cols);

  // Sets the initial threshold with the mean of the total picture
  pair<uchar, uchar> T;
  T.first = (uchar)(OriginalImgStats.Mean + 0.5);
  T.second = 0;

  uchar Rstd = 0;
```

```cpp
  uchar Lstd = 0;
  uchar Rmean = 0;
  uchar Lmean = 0;

  // Iterate till optimum Threshold is found between back- & foreground
  while (T.first != T.second) {
    // Gets an array of the left part of the histogram
    uint32_t i = T.first;
    uint32_t *Left = new uint32_t[i]{};
    while (i-- > 0) {
      Left[i] = OriginalImgStats.bins[i];
    }

    // Gets an array of the right part of the histogram
    uint32_t rightEnd = 256 - T.first;
    uint32_t *Right = new uint32_t[rightEnd]{};
    i = rightEnd;
    while (i-- > 0) {
      Right[i] = OriginalImgStats.bins[i + T.first];
    }

    // Calculate the statistics of both histograms,
    // taking into account the current threshold
    ucharStat_t sLeft(Left, 0, T.first);
    ucharStat_t sRight(Right, T.first, 256);

    // Calculate the new threshold the mean of the means
    T.second = T.first;
    T.first = (uchar)(((sLeft.Mean + sRight.Mean) / 2) + 0.5);

    Rmean = (uchar)(sRight.Mean + 0.5);
    Lmean = (uchar)(sLeft.Mean + 0.5);
    Rstd = (uchar)(sRight.Std + 0.5);
    Lstd = (uchar)(sLeft.Std + 0.5);
    delete[] Left;
    delete[] Right;
  }

  // Assumes the pixel value of the sought object lies between 2 sigma
  int val = 0;
  switch (TypeObject) {
  case Bright:
    val = Rmean - (sigma * Rstd) - thresholdOffset;
    if (val < 0) {
      val = 0;
    } else if (val > 255) {
      val = 255;
    }
    T.first = (uchar)val;
    break;
  case Dark:
    val = Lmean + (sigma * Lstd) + thresholdOffset;
    if (val < 0) {
      val = 0;
    } else if (val > 255) {
      val = 255;
    }
    T.first = (uchar)val;
    break;
  }

  return T.first;
}

/*! Convert a greyscale image to a BW using an automatic Threshold
\param src is the source image as a cv::Mat
\param dst destination image as a cv::Mat
\param TypeObject is an enumerator indicating if the bright or the dark pixels
are the object and should be set to one */
void Segment::ConvertToBW(const Mat &src, Mat &dst, TypeOfObjects Typeobjects) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  LabelledImg.create(OriginalImg.size(), CV_16UC1);
  ConvertToBW(Typeobjects);
  dst = ProcessedImg;
}

/*! Convert a greyscale image to a BW using an automatic Threshold
\param TypeObject is an enumerator indicating if the bright or the dark pixels
are the object and should be set to one         */
void Segment::ConvertToBW(TypeOfObjects Typeobjects) {
  // Determine the threshold
  uchar T = GetThresholdLevel(Typeobjects);
```

```cpp

  // Threshold the picture
  Threshold(T, Typeobjects);
}

/*! Convert a greyscale image to a BW
\param t uchar set the value which is the tiping point
\param TypeObject is an enumerator indicating if the bright or the dark pixels
are the object and should be set to one         */
void Segment::Threshold(uchar t, TypeOfObjects Typeobjects) {
  // Exception handling
  EMPTY_CHECK(OriginalImg);
  CV_Assert(OriginalImg.depth() != sizeof(uchar) ||
            OriginalImg.depth() != sizeof(uint16_t));

  // Create LUT
  uchar LUT_newValue[256]{0};
  if (Typeobjects == Bright) {
    for (uint32_t i = t; i < 256; i++) {
      LUT_newValue[i] = 1;
    }
  } else {
    for (uint32_t i = 0; i <= t; i++) {
      LUT_newValue[i] = 1;
    }
  }

  // Create the pointers to the data
  uchar *P = ProcessedImg.data;
  uchar *O = OriginalImg.data;

  // Fills the ProcessedImg with either a 0 or 1
  for (int i = 0; i < OriginalImg.cols * OriginalImg.rows; i++) {
    P[i] = LUT_newValue[O[i]];
  }
}

/*! Set all the border pixels to a set value
\param *P uchar pointer to the Mat.data
\param setValue uchar the value which is written to the border pixels
*/
void Segment::SetBorder(uchar *P, uchar setValue) {
  // Exception handling
  EMPTY_CHECK(OriginalImg);
  CV_Assert(OriginalImg.depth() != sizeof(uchar) ||
            OriginalImg.depth() != sizeof(uint16_t));

  uint32_t nData = OriginalImg.cols * OriginalImg.rows;

  // Set borderPixels to 2
  uint32_t i = 0;
  uint32_t pEnd = OriginalImg.cols + 1;

  // Set the top row to value 2
  while (i < pEnd) {
    P[i++] = setValue;
  }

  // Set the bottom row to value 2
  i = nData + 1;
  pEnd = nData - OriginalImg.cols;
  while (i-- > pEnd) {
    P[i] = setValue;
  }

  // Sets the first and the last Column to 2
  i = 1;
  pEnd = OriginalImg.rows;
  while (i < pEnd) {
    P[(i * OriginalImg.cols) - 1] = setValue;
    P[(i++ * OriginalImg.cols)] = setValue;
  }
}

/*! Remove the blobs that are connected to the border
\param conn set the pixel connection eight or four
\param chain use the results from the previous operation default value = false;
*/
void Segment::RemoveBorderBlobs(uint32_t border, bool chain) {
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);
  // make Pointers
  uchar *O;
```

```cpp
  CHAIN_PROCESS(chain, O, uchar);
  if (chain) {
    ProcessedImg = TempImg.clone();
  } else {
    ProcessedImg = OriginalImg.clone();
  }

  SHOW_DEBUG_IMG(OriginalImg, uchar, 255, "Original Image RemoverBorderBlobs!",
                 true);
  SHOW_DEBUG_IMG(TempImg, uchar, 255, "Temp Image RemoverBorderBlobs!", true);

  uchar *P = ProcessedImg.data;
  uint32_t cols = ProcessedImg.cols;
  uint32_t rows = ProcessedImg.rows;

  try {
    for (uint32_t i = 0; i < border; i++) {
      for (uint32_t j = 0; j < cols; j++) {
        if (O[(i * cols) + j] == 1 && P[(i * cols) + j] != 2) {
          cv::floodFill(ProcessedImg, cv::Point(j, i), (uchar)2);
        }
      }
    }

    for (uint32_t i = rows - border - 1; i < rows; i++) {
      for (uint32_t j = 0; j < cols; j++) {
        if (O[(i * cols) + j] == 1 && P[(i * cols) + j] != 2) {
          cv::floodFill(ProcessedImg, cv::Point(j, i), (uchar)2);
        }
      }
    }

    for (uint32_t i = border; i < rows - border; i++) {
      for (uint32_t j = 0; j < border; j++) {
        if (O[(i * cols) + j] == 1 && P[(i * cols) + j] != 2) {
          cv::floodFill(ProcessedImg, cv::Point(j, i), (uchar)2);
        }
        if (O[(i * cols) + (cols - j - 1)] == 1 &&
            P[(i * cols) + (cols - j - 1)] != 2) {
          cv::floodFill(ProcessedImg, cv::Point(cols - j - 1, i), (uchar)2);
        }
      }
    }
  } catch (cv::Exception &e) {
  }
  SHOW_DEBUG_IMG(ProcessedImg, uchar, 255,
                 "Processed Image RemoverBorderBlobs before LUT!", true);

  // Change values 2 -> 0
  uchar LUT_newValue[3]{0, 1, 0};
  P = ProcessedImg.data;
  uint32_t nData = rows * cols;
  for (uint32_t i = 0; i < nData; i++) {
    P[i] = LUT_newValue[P[i]];
  }

  SHOW_DEBUG_IMG(ProcessedImg, uchar, 255,
                 "Processed Image RemoverBorderBlobs!", true);
}

/*! Label all the individual blobs in a BW source image. The result are written
to the labelledImg as an ushort
\param conn set the pixel connection eight or four
\param chain use the results from the previous operation default value = false;
\param minBlobArea minimum area when an artifact is considered a blob
*/
void Segment::LabelBlobs(bool chain, uint16_t minBlobArea, Connected conn) {
  // Exception handling
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);

  // make the Pointers to the data
  uchar *O;
  if (chain) {
    TempImg = ProcessedImg.clone();
    ProcessedImg = cv::Mat(OriginalImg.rows, OriginalImg.cols, CV_16UC1);
    O = (uchar *)TempImg.data;
  } else {
    O = (uchar *)OriginalImg.data;
  }
  uint16_t *P = (uint16_t *)LabelledImg.data;

  uint32_t nCols = OriginalImg.cols;
```

```
  uint32_t nRows = OriginalImg.rows;
  uint32_t nData = nCols * nRows;

  vector<vector<uint16_t>> CLdownstream;

  ConnectedBlobs(O, P, CLdownstream, nCols, nRows,
                 conn); // First loop through the image
  SortAdjacencyList(
      CLdownstream); // Sort all the adjacencylists and make unique,

  // identify all the lowest values in the adjacent list
  uint16_t *valueArr = new uint16_t[CLdownstream.size()];
  for (int i = CLdownstream.size() - 1; i >= 0; --i) {
    std::vector<uint16_t *> route;
    uint16_t minVal = i;

    for (uint32_t j = 0; j < CLdownstream[i].size(); j++) {

      // add the first node to the queue;
      route.push_back(&CLdownstream[i][j]);

      // itterate till the last node
      bool lastNodeReached = false;
      while (!lastNodeReached) {
        uint32_t nodesVisited = route.size() - 1;
        if (*route[nodesVisited] < minVal) {
          minVal = *route[nodesVisited];
        }
        route.push_back(&CLdownstream[*route[nodesVisited]][0]);
        if (route[nodesVisited] == route[nodesVisited + 1]) {
          route.pop_back();
          lastNodeReached = true;
        }
      }
      // Set all values to the lowest value
      for (uint32_t k = 0; k < route.size(); k++) {
        *route[k] = minVal;
      }
    }
    valueArr[i] = minVal;
  }

  // Make numbers consecutive
  MakeConsecutive(valueArr, CLdownstream.size(), MaxLabel);

  // Second loop through the pixels to give the values a final value
  for_each(P, P + nData, [&](uint16_t &V) { V = valueArr[V]; });
  delete[] valueArr;
}

/*! Create a BW image with only edges from a BW image
\param src source image as a const cv::Mat
\param dst destination image as a cv::Mat
\param conn set the pixel connection eight or four
\param chain use the results from the previous operation default value = false;
*/
void Segment::GetEdges(const Mat &src, Mat &dst, bool chain, Connected conn) {
  OriginalImg = src;
  GetEdges(chain, conn);
  dst = ProcessedImg;
}

/*! Create a BW image with only edges from a BW image
\param conn set the pixel connection eight or four
\param chain use the results from the previous operation default value = false;
*/
void Segment::GetEdges(bool chain, Connected conn) {
  // Exception handling
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);

  // make Pointers
  uchar *O;
  CHAIN_PROCESS(chain, O, uchar);
  uchar *P = ProcessedImg.data;

  uint32_t nCols = OriginalImg.cols;
  uint32_t nRows = OriginalImg.rows;
  uint32_t nData = nCols * nRows;
  uint32_t pEnd = nData + 1;
  uint32_t i = 0;

  // Loop through the image and set each pixel which has a zero neighbor set it
```

```
  // to two.
  if (conn == Four) {
    // Loop through the picture
    while (i < pEnd) {
      // If current value = zero processed value = zero
      if (O[i] == 0) {
        P[i] = 0;
      }
      // If current value = 1 check North West, South and East and act
      // accordingly
      else if (O[i] == 1) {
        uchar *nPixels = new uchar[4];
        nPixels[0] = O[i - 1];
        nPixels[1] = O[i - nCols];
        nPixels[2] = O[i + 1];
        nPixels[3] = O[i + nCols];

        // Sort the neighbors for easier checking
        SoilMath::Sort::QuickSort<uchar>(nPixels, 4);
        if (nPixels[0] == 0) {
          P[i] = 1;
        } else {
          P[i] = 0;
        }
      } else {
        throw Exception::PixelValueOutOfBoundException();
      }
      i++;
    }
  } else {
    // Loop through the picture
    while (i < pEnd) {
      // If current value = zero processed value = zero
      if (O[i] == 0) {
        P[i] = 0;
      }
      // If current value = 1 check North West, South and East and act
      // accordingly
      else if (O[i] == 1) {
        uchar *nPixels = new uchar[8];
        nPixels[0] = O[i - 1];
        nPixels[1] = O[i - nCols];
        nPixels[2] = O[i - nCols - 1];
        nPixels[3] = O[i - nCols + 1];
        nPixels[4] = O[i + 1];
        nPixels[5] = O[i + nCols + 1];
        nPixels[6] = O[i + nCols];
        nPixels[7] = O[i + nCols - 1];

        // Sort the neighbors for easier checking
        SoilMath::Sort::QuickSort<uchar>(nPixels, 8);

        if (nPixels[0] == 0) {
          P[i] = 1;
        } else {
          P[i] = 0;
        }
      } else {
        throw Exception::PixelValueOutOfBoundException();
      }
      i++;
    }
  }
}

void Segment::GetEdgesEroding(bool chain) {
  // Exception handling
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);

  // make Pointers
  uchar *O;
  CHAIN_PROCESS(chain, O, uchar);
  uchar *P = ProcessedImg.data;

  uint32_t nCols = OriginalImg.cols;
  uint32_t nRows = OriginalImg.rows;
  uint32_t nData = nCols * nRows;

  // Setup the erosion
  MorphologicalFilter eroder;
  if (chain) {
    eroder.OriginalImg = TempImg;
```

```
  } else {
    eroder.OriginalImg = OriginalImg;
  }
  // Setup the processed image of the eroder
  eroder.ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  eroder.ProcessedImg.setTo(0);
  // Setup the mask
  Mat mask(3, 3, CV_8UC1, 1);
  // Erode the image
  eroder.Erosion(mask, false);

  // Loop through the image and set the not eroded pixels to zero
  for (uint32_t i = 0; i < nData; i++) {
    if (O[i] != eroder.ProcessedImg.data[i]) {
      P[i] = 1;
    } else {
      P[i] = 0;
    }
  }

  // ProcessedImg = OriginalImg.clone() - eroder.ProcessedImg.clone();

  SHOW_DEBUG_IMG(eroder.ProcessedImg, uchar, 255, "Eroded img Processed Image!",
                 true);
  SHOW_DEBUG_IMG(ProcessedImg, uchar, 255, "GetEdgesEroding Processed Image!",
                 true);
}

/*! Create a BlobList subtracting each individual blob out of a Labelled image.
If the labelled image is empty build a new one with a BW image.
\param conn set the pixel connection eight or four
\param chain use the results from the previous operation default value = false;
*/
void Segment::GetBlobList(bool chain, Connected conn) {
  // Exception handling
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);

  // If there isn't a labelledImg make one
  if (MaxLabel < 1) {
    LabelBlobs(chain, 5, conn);
  }

  // Make an empty BlobList
  uint32_t nCols = OriginalImg.cols;
  uint32_t nRows = OriginalImg.rows;
  uint32_t nData = nCols * nRows;
  RectList_t rectList;

  // Calculate Stats the statistics
  uint16Stat_t LabelStats((uint16_t *)LabelledImg.data, LabelledImg.cols,
                          LabelledImg.rows, MaxLabel + 1, 0, MaxLabel);

  BlobList.reserve(LabelStats.EndBin);
  rectList.reserve(LabelStats.EndBin);

  BlobList.push_back(Blob_t(0, 0));
  rectList.push_back(Rect_t(0, 0, 0, 0));

  for (uint32_t i = 1; i < LabelStats.EndBin; i++) {
    BlobList.push_back(Blob_t(i, LabelStats.bins[i]));
    rectList.push_back(Rect_t(nCols, nRows, 0, 0));
  }

  // make Pointers
  uint16_t *L = (uint16_t *)LabelledImg.data;

  uint32_t currentX, currentY;
  // uint16_t leftX, leftY, rightX, rightY;
  // Loop through the labeled image and extract the Blobs
  for (uint32_t i = 0; i < nData; i++) {
    if (L[i] != 0) {
      /* Determine the current x and y value of the current blob and
      checks if it is min/max */
      currentY = i / nCols;
      currentX = i % nCols;

      // Min value
      if (currentX < rectList[L[i]].leftX) {
        rectList[L[i]].leftX = currentX;
      }
      if (currentY < rectList[L[i]].leftY) {
        rectList[L[i]].leftY = currentY;
```

```
      }

      // Max value
      if (currentX > rectList[L[i]].rightX) {
        rectList[L[i]].rightX = currentX;
      }
      if (currentY > rectList[L[i]].rightY) {
        rectList[L[i]].rightY = currentY;
      }
    }
  }

  // Loop through the BlobList and finalize it
  uint8_t *LUT_filter = new uint8_t[MaxLabel + 1]{};
  for (uint32_t i = 1; i <= MaxLabel; i++) {
    LUT_filter[i] = 1;
    BlobList[i].ROI.y = rectList[i].leftY;
    BlobList[i].ROI.x = rectList[i].leftX;
    BlobList[i].ROI.height = rectList[i].rightY - rectList[i].leftY + 1;
    BlobList[i].ROI.width = rectList[i].rightX - rectList[i].leftX + 1;
    BlobList[i].Img = CopyMat<uint8_t, uint16_t>(
        LabelledImg(BlobList[i].ROI).clone(), LUT_filter, CV_8UC1);
    //SHOW_DEBUG_IMG(BlobList[i].Img, uchar, 255, "Blob", true);
    LUT_filter[i] = 0;
  }
  delete[] LUT_filter;

  // Remove background blob
  BlobList.erase(BlobList.begin());
}

void Segment::FillHoles(bool chain) {
  // Exception handling
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);

  // make Pointers
  uchar *O;
  CHAIN_PROCESS(chain, O, uchar);
  if (chain) {
    ProcessedImg = TempImg.clone();
  } else {
    ProcessedImg = OriginalImg.clone();
  }

  uchar *P = ProcessedImg.data;

  // Determine the starting point of the floodfill
  int itt = -1;
  while (P[++itt] != 0)
    ;
  uint16_t row = static_cast<uint16_t>(itt / OriginalImg.rows);
  uint16_t col = static_cast<uint16_t>(itt % OriginalImg.rows);

  // Fill the outside
  // FloodFill(O, P, row, col, 2, 0);

  try {
    cv::floodFill(ProcessedImg, cv::Point(col, row), cv::Scalar(2));
  } catch (cv::Exception &e) {
  }

  // Set the unreached areas to 1 and the outside to 0;
  uchar LUT_newVal[3] = {1, 1, 0};
  uint32_t nData = OriginalImg.rows * OriginalImg.cols;
  uint32_t i = 0;
  while (i <= nData) {
    P[i] = LUT_newVal[P[i]];
    i++;
  }
}

/*!
 * \brief Segment::FloodFill
 * \param O
 * \param P
 * \param row
 * \param col
 * \param fillValue
 * \param OldValue
 */
void Segment::FloodFill(uchar *O, uchar *P, uint16_t row, uint16_t col,
                        uchar fillValue, uchar OldValue) {
```

```
  if (row < 0 || row > OriginalImg.rows) {
    return;
  }
  if (col < 0 || col > OriginalImg.cols) {
    return;
  }
  if (P[col + row * OriginalImg.rows] == OldValue) {
    P[col + row * OriginalImg.rows] = fillValue;
    FloodFill(O, P, row + 1, col, fillValue, OldValue);
    FloodFill(O, P, row, col + 1, fillValue, OldValue);
    FloodFill(O, P, row - 1, col, fillValue, OldValue);
    FloodFill(O, P, row, col - 1, fillValue, OldValue);
  }
}

/*!
 * \brief Segment::SortAdjacencyList Sort the the sub vectors
 * \param adj std::vector<std::vector<uint16_t>> &adj
 */
void Segment::SortAdjacencyList(std::vector<std::vector<uint16_t>> &adj) {
  uint32_t j = 0;
  for_each(adj.begin(), adj.end(), [&](std::vector<uint16_t> &L) {
    std::sort(L.begin(), L.end());
    std::vector<uint16_t>::iterator it;
    it = std::unique(L.begin(), L.end());
    L.resize(std::distance(L.begin(), it));
    if (L.size() > 1) {
      for (std::vector<uint16_t>::iterator iter = L.begin(); iter != L.end();
           ++iter) {
        if (*iter == j) {
          L.erase(iter);
          break;
        }
      }
    }
    j++;
  });
}

/*!
 * \brief Segment::ConnectedBlobs Connect all the blobs and created the
 * adjacency list
 * \param O
 * \param P
 * \param adj
 * \param nCols
 * \param nRows
 * \param conn
 */
void Segment::ConnectedBlobs(uchar *O, uint16_t *P,
                             std::vector<std::vector<uint16_t>> &adj,
                             uint32_t nCols, uint32_t nRows, Connected conn) {
  // Determine the size of the array for beginning and endrow and middle of a
  // row
  uint32_t noConn[3] = {static_cast<uint32_t>(conn),
                        (static_cast<uint32_t>(conn) / 2),
                        (static_cast<uint32_t>(conn) / 2) + 1};
  uint32_t lastConn[3] = {noConn[0] - 1, noConn[1] - 1, noConn[2] - 1};
  uint32_t nData = nCols * nRows;

  uint16_t currentlbl = 0;
  vector<uint16_t> zeroVector;
  zeroVector.push_back(currentlbl);
  adj.push_back(zeroVector);

  // Determine which borderpixels should be handled differently
  uchar *nRow = new uchar[nData]{};
  for (uint32_t i = nCols; i < nData; i += nCols) {
    nRow[i] = 1;
    nRow[i - 1] = 2;
  }

  // Set the first pixel
  if (O[0] == 0) {
    P[0] = 0;
  } else if (O[0] == 1) {
    P[0] = 1;
  } else {
    throw Exception::PixelValueOutOfBoundException();
  }

  // Walk through the toprow and determine if it’s a new blob or it’s connected
  // with previously determine blob
```

```
  for (uint32_t i = 1; i < nCols; i++) {
    if (O[i] == 0) {
      P[i] = 0;
    } else if (O[i] == 1) {
      // If West is zero assume this is a new blob
      if (P[i - 1] == 0) {
        P[i] = ++currentlbl;
        vector<uint16_t> cVector;
        cVector.push_back(currentlbl);
        adj.push_back(cVector);
      } else { // set as previous blob
        P[i] = P[i - 1];
      }
    } else { // Value of of bounds
      throw Exception::PixelValueOutOfBoundException();
    }
  }

  // walk through each pixel and determine if it's a new blob or it's connected
  // with previously determine blob
  for (uint32_t i = OriginalImg.cols; i < nData; i++) {
    if (O[i] == 0) { // Original pixel = 0
      P[i] = 0;
    } else if (O[i] == 1) {
      // Get an array of Neighboring Pixels
      uint16_t *nPixels = new uint16_t[noConn[nRow[i]]];
      if (nRow[i] != 1) {
        nPixels[0] = P[i - 1];
      }
      uint32_t j = i - nCols - ((nRow[i] == 1) ? 0 : ((conn == Four) ? 0 : 1));
      for_each(nPixels + ((nRow[i] != 1) ? 1 : 0), nPixels + noConn[nRow[i]],
               [&](uint16_t &N) { N = P[j++]; });

      // Sort the neighbors for easier checking
      SoilMath::Sort::QuickSort<uint16_t>(nPixels, noConn[nRow[i]]);

      // If all are zero assume this is a new blob
      if (nPixels[lastConn[nRow[i]]] == 0) {
        P[i] = ++currentlbl;
        vector<uint16_t> cVector;
        cVector.push_back(currentlbl);
        adj.push_back(cVector);
      } else {
        /* Sets the processed value to the smallest non-zero value and update
         * the connectedLabels */
        for (uint32_t j = 0; j < noConn[nRow[i]]; j++) {
          if (nPixels[j] > 0) {
            P[i] = nPixels[j];
            break;
          }
        }

        /* If previous blobs belong to different connected components set the
         * current processed value to the lowest value and remember that the
         * other values should be the lowest value*/
        if (P[i] != nPixels[lastConn[nRow[i]]]) {
          for (int j = lastConn[nRow[i]]; j >= 0; --j) {
            if (nPixels[j] <= P[i]) {
              break;
            } else {
              adj[nPixels[j]].push_back(P[i]);
            }
          }
        }
      }
      delete[] nPixels;
    } else {
      throw Exception::PixelValueOutOfBoundException();
    }
  }
  delete[] nRow;
}

/*!
 * \brief Segment::InvertAdjacencyList invert the adjecencylist for upstream
 * (unused)
 * \param adj
 * \param adjInv
 */
void Segment::InvertAdjacencyList(std::vector<std::vector<uint16_t>> &adj,
                                  std::vector<std::vector<uint16_t>> &adjInv) {
  // Build the inverted vector
  adjInv.resize(adj.size());
```

```cpp
  uint16_t count = 0;
  for_each(adj.begin(), adj.end(), [&](std::vector<uint16_t> &V) {
    for_each(V.begin(), V.end(),
             [&](uint16_t &C) { adjInv[C].push_back(count); });
    count++;
  });
}

/*!
 * \brief Segment::MakeConsecutive make the valueArr consequative numbers
 * \param valueArr
 * \param noElem
 * \param maxLabel
 */
void Segment::MakeConsecutive(uint16_t *valueArr, uint32_t noElem,
                              uint16_t &maxLabel) {
  std::vector<std::vector<uint16_t>> conseq;
  conseq.resize(noElem);
  for (uint32_t i = 0; i < noElem; i++) {
    conseq[valueArr[i]].push_back(i);
  }
  uint32_t count = 1;
  for (uint32_t i = 1; i < noElem; i++) {
    if (conseq[i].size() > 0) {
      for (uint32_t j = 0; j < conseq[i].size(); j++) {
        valueArr[conseq[i][j]] = count;
      }
      count++;
    }
  }
  maxLabel = count - 1;
}

/*!
 * \brief Segment::MakeConsecutive probably a fault in this function. Don’t use
 * \param valueArr
 * \param keyArr
 * \param noElem
 * \param maxlabel
 */
void Segment::MakeConsecutive(uint16_t *valueArr, uint16_t *keyArr,
                              uint16_t noElem, uint16_t &maxlabel) {
  SoilMath::Sort::QuickSort<uint16_t>(valueArr, keyArr, noElem);
  uint16_t count = 0;
  for (uint32_t i = 1; i < noElem; i++) {
    if (valueArr[i] != valueArr[i - 1]) {
      count++;
    }
    valueArr[i] = count;
  }
  SoilMath::Sort::QuickSort<uint16_t>(keyArr, valueArr, noElem);
  delete[] keyArr;
  maxlabel = count;
}
}
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! Collection header of all the basic Vision headers*/

#pragma once
#include "Conversion.h"
#include "Enhance.h"
#include "Segment.h"
#include "MorphologicalFilter.h"
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
// Debuging helper macros
#ifndef DEBUG
#define DEBUG
#endif

#ifdef DEBUG
#include <limits>
#include <opencv2/highgui/highgui.hpp>
#include <vector>
#include "ImageProcessing.h"
#ifndef SHOW_DEBUG_IMG
#define SHOW_DEBUG_IMG(img, T1, maxVal, windowName, scale)                      \
  Vision::ImageProcessing::ShowDebugImg<T1>(img, maxVal, windowName, scale)
#endif // !SHOW_DEBUG_IMG
#else
#ifndef SHOW_DEBUG_IMG
#define SHOW_DEBUG_IMG(img, T1, maxVal, windowName, scale)
#endif // !SHOW_DEBUG_IMG
#endif
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class PixelValueOutOfBoundException
Exception class which is thrown when an unexpected pixel value has to be
computed
*/
#pragma once

#include <exception>
#include <string>

using namespace std;

namespace Vision {
namespace Exception {
class PixelValueOutOfBoundException : public std::exception {
public:
  PixelValueOutOfBoundException(string m = "Current pixel value out of bounds!")
      : msg(m){};
  ~PixelValueOutOfBoundException() _GLIBCXX_USE_NOEXCEPT{};
  const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };

private:
  string msg;
};
}
}
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class EmtpyImageException
Exception class which is thrown when operations are about to start on a empty
image.
*/

#pragma once

#include <exception>
#include <string>

using namespace std;

namespace Vision {
namespace Exception {
class EmtpyImageException : public std::exception {
public:
  EmtpyImageException(string m = "Empty Image!") : msg(m){};
  ~EmtpyImageException() _GLIBCXX_USE_NOEXCEPT{};
  const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };

private:
  string msg;
};
}
}
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
#define MORPHOLOGICALFILTER_VERSION 1

#include "ImageProcessing.h"

namespace Vision {
class MorphologicalFilter : public ImageProcessing {
public:
  enum FilterType { OPEN, CLOSE, ERODE, DILATE, NONE };

  MorphologicalFilter();
  MorphologicalFilter(FilterType filtertype);
  MorphologicalFilter(const Mat &src, FilterType filtertype = FilterType::NONE);
  MorphologicalFilter(const MorphologicalFilter &rhs);

  ~MorphologicalFilter();

  MorphologicalFilter &operator=(MorphologicalFilter &rhs);

  void Dilation(const Mat &mask, bool chain = false);
  void Erosion(const Mat &mask, bool chain = false);

  void Close(const Mat &mask, bool chain = false);
  void Open(const Mat &mask, bool chain = false);

private:
  void Filter(const Mat &mask, bool chain, uchar startVal, uchar newVal,
              uchar switchVal);
};
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#include "MorphologicalFilter.h"

namespace Vision {
MorphologicalFilter::MorphologicalFilter() {}

MorphologicalFilter::MorphologicalFilter(FilterType filtertype) {
  switch (filtertype) {
  case FilterType::OPEN:
    Open(OriginalImg);
    break;
  case FilterType::CLOSE:
    Close(OriginalImg);
    break;
  case FilterType::ERODE:
    Erosion(OriginalImg);
    break;
  case FilterType::DILATE:
    Dilation(OriginalImg);
    break;
  case FilterType::NONE:
    break;
  }
}

MorphologicalFilter::MorphologicalFilter(const Mat &src,
                                         FilterType filtertype) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  switch (filtertype) {
  case FilterType::OPEN:
    Open(OriginalImg);
    break;
  case FilterType::CLOSE:
    Close(OriginalImg);
    break;
  case FilterType::ERODE:
    Erosion(OriginalImg);
    break;
  case FilterType::DILATE:
    Dilation(OriginalImg);
    break;
  case FilterType::NONE:
    break;
  }
}

MorphologicalFilter::MorphologicalFilter(const MorphologicalFilter &rhs) {
  this->OriginalImg = rhs.OriginalImg;
  this->ProcessedImg = rhs.ProcessedImg;
  this->TempImg = rhs.ProcessedImg;
}

MorphologicalFilter::~MorphologicalFilter() {}

MorphologicalFilter &MorphologicalFilter::operator=(MorphologicalFilter &rhs) {
  if (&rhs != this) {
    this->OriginalImg = rhs.OriginalImg;
    this->ProcessedImg = rhs.ProcessedImg;
    this->TempImg = rhs.TempImg;
  }
  return *this;
}

void MorphologicalFilter::Open(const Mat &mask, bool chain) {
  Erosion(mask, chain);
  Dilation(mask, true);
}

void MorphologicalFilter::Close(const Mat &mask, bool chain) {
  Dilation(mask, chain);
  Erosion(mask, true);
}

void MorphologicalFilter::Dilation(const Mat &mask, bool chain) {
  Filter(mask, chain, 0, 1, 1);
}
```

```cpp

void MorphologicalFilter::Erosion(const Mat &mask, bool chain) {
  Filter(mask, chain, 1, 0, 0);
}

void MorphologicalFilter::Filter(const Mat &mask, bool chain, uchar startVal,
                                 uchar newVal, uchar switchVal) {
  // Exception handling
  CV_Assert(OriginalImg.depth() != sizeof(uchar));
  EMPTY_CHECK(OriginalImg);
  if (mask.cols % 2 == 0 || mask.cols < 3) {
    throw Exception::WrongKernelSizeException("Wrong Kernelsize columns!");
  }
  if (mask.rows % 2 == 0 || mask.rows < 3) {
    throw Exception::WrongKernelSizeException("Wrong Kernelsize rows!");
  }

  uint32_t hKsizeCol = (mask.cols / 2);
  uint32_t hKsizeRow = (mask.rows / 2);

  // make Pointers
  Mat workOrigImg(ProcessedImg.rows + mask.rows, ProcessedImg.cols + mask.cols,
                  CV_8UC1);
  workOrigImg.setTo(0);
  if (chain) {
    ProcessedImg.copyTo(workOrigImg(
        cv::Rect(hKsizeCol, hKsizeRow, ProcessedImg.cols, ProcessedImg.rows)));
    // workOrigImg(cv::Rect(hKsizeCol, hKsizeRow, ProcessedImg.cols,
    // ProcessedImg.rows)) = ProcessedImg.clone();
  } else {
    OriginalImg.copyTo(workOrigImg(
        cv::Rect(hKsizeCol, hKsizeRow, ProcessedImg.cols, ProcessedImg.rows)));
    // workOrigImg(cv::Rect(hKsizeCol, hKsizeRow, ProcessedImg.cols,
    // ProcessedImg.rows)) = OriginalImg.clone();
  }
  uchar *O = workOrigImg.data;

  Mat workProcImg(ProcessedImg.rows + mask.rows, ProcessedImg.cols + mask.cols,
                  CV_8UC1);
  uchar *P = workProcImg.data;

  // Init the relevant data
  //uint32_t nData = OriginalImg.cols * OriginalImg.rows;
  uint32_t nWData = workProcImg.cols * workProcImg.rows;
  uint32_t nWStart = (hKsizeRow * workProcImg.cols) + hKsizeRow;
  uint32_t nWEnd = nWData - hKsizeCol - hKsizeRow * workProcImg.cols - 1;
  uchar *nRow = GetNRow(nWData, hKsizeCol, workProcImg.cols, workProcImg.rows);
  int MaskPixel = 0, OPixel = 0;

  workProcImg.setTo(0);
  if (startVal != 0) {
    workProcImg(cv::Rect(hKsizeCol, hKsizeRow, ProcessedImg.cols,
                         ProcessedImg.rows)).setTo(startVal);
  }
  SHOW_DEBUG_IMG(workOrigImg, uchar, 255, "workOrigImg Filter!", false);
  SHOW_DEBUG_IMG(mask, uchar, 255, "Filter mask", true);

  for (uint32_t i = nWStart; i < nWEnd; i++) {
    // Checks if pixel isn't a border pixel and progresses to the new row
    if (nRow[i] == 1) {
      i += mask.cols;
    }
    for (int r = 0; r < mask.rows; r++) {
      for (int c = 0; c < mask.cols; c++) {
        MaskPixel = c + r * mask.cols;
        OPixel = i - hKsizeCol + c + (r - hKsizeRow) * workProcImg.cols;
        if (mask.data[MaskPixel] == 1 && O[OPixel] == switchVal) {
          P[i] = newVal;
          c = mask.cols;
          r = mask.rows;
        }
      }
    }
  }
  delete[] nRow;
  SHOW_DEBUG_IMG(workProcImg, uchar, 255, "workProcImg Filter!", true);
  ProcessedImg = workProcImg(Rect(hKsizeCol, hKsizeRow, ProcessedImg.cols,
                                  ProcessedImg.rows)).clone();
  SHOW_DEBUG_IMG(ProcessedImg, uchar, 255, "Processed Image Filter!", true);
}
}
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
/*! Current class version*/
#define IMAGEPROCESSING_VERSION 1

/*! MACRO which sets the original pointer to the original image or a clone of
 * the earlier processed image */
#define CHAIN_PROCESS(chain, O, type)                                         \
  if (chain) {                                                                \
    TempImg = ProcessedImg.clone();                                           \
    O = (type *)TempImg.data;                                                 \
  } else {                                                                    \
    O = (type *)OriginalImg.data;                                             \
  }
/*! MACRO which trows an EmtpyImageException if the matrix is empty*/
#define EMPTY_CHECK(img)                                                      \
  if (img.empty()) {                                                          \
    throw Exception::EmtpyImageException();                                   \
  }

#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>

#include <stdint.h>
#include <cmath>
#include <vector>
#include <string>

#include <boost/signals2.hpp>
#include <boost/bind.hpp>

#include "EmptyImageException.h"
#include "WrongKernelSizeException.h"
#include "ChannelMismatchException.h"
#include "PixelValueOutOfBoundException.h"
#include "VisionDebug.h"

using namespace cv;

namespace Vision {
class ImageProcessing {
public:
  typedef boost::signals2::signal<void(float, std::string)> Progress_t;
  boost::signals2::connection
  connect_Progress(const Progress_t::slot_type &subscriber);

protected:
  uchar *GetNRow(int nData, int hKsize, int nCols, uint32_t totalRows);
  Mat TempImg;

  Progress_t prog_sig;

public:
  ImageProcessing();
  ~ImageProcessing();
  Mat OriginalImg;
  Mat ProcessedImg;

  double currentProg = 0.;
  double ProgStep = 0.;

  static std::vector<Mat> extractChannel(const Mat &src);

  /*! Copy a matrix to a new matrix with a LUT mask
  \param src the source image
  \param *LUT type T with a LUT to filter out unwanted pixel values
  \param cvType an in where you can pas CV_UC8C1 etc.
  \return The new matrix
  */
  template <typename T1, typename T2>
  static Mat CopyMat(const Mat &src, T1 *LUT, int cvType) {
    Mat dst(src.size(), cvType);
    uint32_t nData = src.rows * src.cols * dst.step[1];
    if (cvType == 0 || cvType == 8 || cvType == 16 || cvType == 24) {
      for (uint32_t i = 0; i < nData; i += dst.step[1]) {
        dst.data[i] =
            static_cast<uint8_t>(LUT[*(T2 *)(src.data + (i * src.step[1]))]);
```

```
      }
    } else if (cvType == 1 || cvType == 9 || cvType == 17 || cvType == 25) {
      for (uint32_t i = 0; i < nData; i += src.step[1]) {
        dst.data[i] =
            static_cast<int8_t>(LUT[*(T2 *)(src.data + (i * src.step[1]))]);
      }
    } else if (cvType == 2 || cvType == 10 || cvType == 18 || cvType == 26) {
      for (uint32_t i = 0; i < nData; i += src.step[1]) {
        dst.data[i] =
            static_cast<uint16_t>(LUT[*(T2 *)(src.data + (i * src.step[1]))]);
      }
    } else if (cvType == 3 || cvType == 11 || cvType == 19 || cvType == 27) {
      for (uint32_t i = 0; i < nData; i += src.step[1]) {
        dst.data[i] =
            static_cast<int16_t>(LUT[*(T2 *)(src.data + (i * src.step[1]))]);
      }
    } else if (cvType == 4 || cvType == 12 || cvType == 20 || cvType == 28) {
      for (uint32_t i = 0; i < nData; i += src.step[1]) {
        dst.data[i] =
            static_cast<int32_t>(LUT[*(T2 *)(src.data + (i * src.step[1]))]);
      }
    }
    return dst;
  }

  /*! Copy a matrix to a new matrix with a mask
  \param src the source image
  \param *LUT type T with a LUT to filter out unwanted pixel values
  \param cvType an in where you can pas CV_UC8C1 etc.
  \return The new matrix
  */
  template <typename T1>
  static Mat CopyMat(const Mat &src, const Mat &mask, int cvType) {
    if (src.size != mask.size) {
      throw Exception::WrongKernelSizeException(
          "Mask not the same size as src Exception!");
    }
    if (mask.channels() != 1) {
      throw Exception::WrongKernelSizeException(
          "Mask has more then 1 channel Exception!");
    }
    Mat dst(src.size(), cvType);

    vector<Mat> exSrc = Vision::ImageProcessing::extractChannel(src);
    vector<Mat> exDst;

    int cvBaseType = cvType % 8;
    for_each(exSrc.begin(), exSrc.end(), [&](const Mat &sItem) {
      Mat dItem(src.size(), cvBaseType);
      std::transform(sItem.begin<T1>(), sItem.end<T1>(), mask.begin<T1>(),
                     dItem.begin<T1>(),
                     [](const T1 &s, const T1 &m) -> T1 { return s * m; });
      exDst.push_back(dItem);
    });

    merge(exDst, dst);

    return dst;
  }

  template <typename T1>
  static void ShowDebugImg(cv::Mat img, T1 maxVal, std::string windowName,
                           bool scale = true) {
    if (img.rows > 0 && img.cols > 0) {
      cv::Mat tempImg(img.size(), img.type());
      if (scale == true) {
        std::vector<cv::Mat> exSrc = extractChannel(img);
        std::vector<cv::Mat> exDst;
        int cvBaseType = img.type() % 8;
        T1 MatMin = std::numeric_limits<T1>::max();
        T1 MatMax = std::numeric_limits<T1>::min();

        // Find the global max and min
        for_each(exSrc.begin(), exSrc.end(), [&](const Mat &sItem) {
          std::for_each(sItem.begin<T1>(), sItem.end<T1>(), [&](const T1 &s) {
            if (s > MatMax) {
              MatMax = s;
            } else if (s < MatMin) {
              MatMin = s;
            }
          });
        });

```

```
        int Range = MatMax - MatMin;
        if (Range < 1)
          Range = maxVal;

        // Convert the values
        for_each(exSrc.begin(), exSrc.end(), [&](const cv::Mat &sItem) {
          Mat dItem(img.size(), cvBaseType);
          std::transform(sItem.begin<T1>(), sItem.end<T1>(), dItem.begin<T1>(),
                         [&](const T1 &s) -> T1 {
                           return (T1)round(((s - MatMin) * maxVal) / Range);
                         });
          exDst.push_back(dItem);
        });

        merge(exDst, tempImg);
      } else {
        tempImg = img;
      }
      cv::namedWindow(windowName, cv::WINDOW_NORMAL);
      cv::imshow(windowName, tempImg);
      cv::waitKey(0);
      cv::destroyWindow(windowName);
    };
  };
};
}
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once

#include <vector>
#include <queue>
#include <string>
#include <stdint.h>
#include <iostream>
#include <algorithm>
#include <utility>

#include <boost/range/adaptor/reversed.hpp>

#include "opencv2/imgproc/imgproc.hpp"

#include "ImageProcessing.h"
#include "MorphologicalFilter.h"
#include "../SoilMath/SoilMath.h"

namespace Vision {
class Segment : public ImageProcessing {
public:
  /*! Coordinates for the region of interest*/
  typedef struct Rect {
    uint16_t leftX;  /*!< Left X coordinate*/
    uint16_t leftY;  /*!< Left Y coordinate*/
    uint16_t rightX; /*!< Right X coordinate*/
    uint16_t rightY; /*!< Right Y coordinate*/
    Rect(uint16_t lx, uint16_t ly, uint16_t rx, uint16_t ry)
        : leftX(lx), leftY(ly), rightX(rx), rightY(ry){};
  } Rect_t;

  typedef std::vector<Vision::Segment::Rect_t> RectList_t;

  /*! Individual blob*/
  typedef struct Blob {
    uint16_t Label; /*!< ID of the blob*/
    cv::Mat Img;   /*!< BW image of the blob all the pixel belonging to the blob
                      are set to 1 others are 0*/
    cv::Rect ROI;  /*!< Coordinates for the blob in the original picture as a
                      cv::Rect*/
    uint32_t Area; /*!< Calculated stats of the blob*/
    Blob(uint16_t label, uint32_t area) : Label(label), Area(area){};
  } Blob_t;

  typedef std::vector<Blob_t> BlobList_t;
  BlobList_t BlobList; /*!< vector with all the individual blobs*/

  /*! Enumerator to indicate what kind of object to extract  */
  enum TypeOfObjects {
    Bright, /*!< Enum value Bright object */
    Dark    /*!< Enum value Dark object. */
  };

  /*! Enumerator to indicate how the pixel correlate between each other in a
   * blob*/
  enum Connected {
    Four =
        2, /*!< Enum Four connected, relation between Center, North, East, South
          and West*/
    Eight =
        4 /*!< Enum Eight connected, relation between Center, North, NorthEast,
         East, SouthEast, South, SouthWest, West and NorthWest */
  };

  /*!< Enumerator which indicate which Segmentation technique should be used */
  enum SegmentationType {
    Normal, /*!< Segmentation looking at the intensity of an individual pixel */
    LabNeuralNet, /*!< Segmentation looking at the chromatic a* and b* of the
                     processed pixel and it's surrounding pixels, feeding it in
                     an Neural Net */
    GraphMinCut /*!< Segmentation using a graph function and the minimum cut */
  };

  cv::Mat LabelledImg;   /*!< Image with each individual blob labeled with a
                            individual number */
  uint16_t MaxLabel = 0; /*!< Maximum labels found in the labelled image*/
```

```
  uint16_t noOfFilteredBlobs =
      0; /*!< Total numbers of blobs that where filtered beacuse the where
            smaller than the minBlobArea*/

  ucharStat_t OriginalImgStats; /*!< Statistical data from the original image*/
  uint8_t ThresholdLevel = 0;   /*!< Current calculated threshold level*/

  float sigma = 2;
  uint32_t thresholdOffset = 4;

  Segment();
  Segment(const Mat &src);
  Segment(const Segment &rhs);

  ~Segment();

  Segment &operator=(Segment &rhs);

  void LoadOriginalImg(const Mat &src);

  void ConvertToBW(TypeOfObjects Typeobjects);
  void ConvertToBW(const Mat &src, Mat &dst, TypeOfObjects Typeobjects);

  void GetEdges(bool chain = false, Connected conn = Eight);
  void GetEdges(const Mat &src, Mat &dst, bool chain = false,
                Connected conn = Eight);

  void GetEdgesEroding(bool chain = false);

  void GetBlobList(bool chain = false, Connected conn = Eight);

  void Threshold(uchar t, TypeOfObjects Typeobjects);

  void LabelBlobs(bool chain = false, uint16_t minBlobArea = 25,
                  Connected conn = Eight);

  void RemoveBorderBlobs(uint32_t border = 1, bool chain = false);

  void FillHoles(bool chain = false);

private:
  uint8_t GetThresholdLevel(TypeOfObjects TypeObject);
  void SetBorder(uchar *P, uchar setValue);
  void FloodFill(uchar *O, uchar *P, uint16_t x, uint16_t y, uchar fillValue,
                 uchar OldValue);
  void MakeConsecutive(uint16_t *valueArr, uint32_t noElem, uint16_t &maxlabel);
  void MakeConsecutive(uint16_t *valueArr, uint16_t *keyArr, uint16_t noElem,
                       uint16_t &maxlabel);
  void SortAdjacencyList(std::vector<std::vector<uint16_t>> &adj);
  void ConnectedBlobs(uchar *O, uint16_t *P,
                      std::vector<std::vector<uint16_t>> &adj, uint32_t nCols,
                      uint32_t nRows, Connected conn);
  void InvertAdjacencyList(std::vector<std::vector<uint16_t>> &adj,
                           std::vector<std::vector<uint16_t>> &adjInv);
};
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class Enhance
class which enhances a greyscale cv::Mat image
*/
#include "Enhance.h"

namespace Vision {
/*! Constructor*/
Enhance::Enhance() {}

/*! Constructor
\param src cv::Mat source image
*/
Enhance::Enhance(const Mat &src) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
}

Enhance::Enhance(const Enhance &rhs) {
  this->OriginalImg = rhs.OriginalImg;
  this->ProcessedImg = rhs.OriginalImg;
  this->TempImg = rhs.TempImg;
}

/*! Constructor
\param src cv::Mat source image
\param dst cv::Mat destination image
\param kernelsize an uchar which represent the kernelsize should be an uneven
number higher than two
\param factor float which indicates the amount the effect should take place
standard value is 1.0 only used in the adaptive contrast stretch enhancement
\param operation enumerator EnhanceOperation which enhancement should be
performed
*/
Enhance::Enhance(const Mat &src, Mat &dst, uchar kernelsize, float factor,
                 EnhanceOperation operation) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  switch (operation) {
  case Vision::Enhance::_AdaptiveContrastStretch:
    AdaptiveContrastStretch(kernelsize, factor);
    break;
  case Vision::Enhance::_Blur:
    Blur(kernelsize);
    break;
  case Vision::Enhance::_HistogramEqualization:
    HistogramEqualization();
    break;
  }
  dst = ProcessedImg;
}

/*! Dec-constructor*/
Enhance::~Enhance() {}

Enhance &Enhance::operator=(Enhance rhs) {
  if (&rhs != this) {
    this->OriginalImg = rhs.OriginalImg;
    this->ProcessedImg = rhs.ProcessedImg;
    this->TempImg = rhs.ProcessedImg;
  }
  return *this;
}

/*! Calculate the standard deviation of the neighboring pixels
\param O uchar pointer to the current pixel of the original image
\param i current counter
\param hKsize half the kernelsize
\param nCols total number of columns
\param noNeighboursPix total number of neighboring pixels
\param mean mean value of the neighboring pixels
\return standard deviation
*/
float Enhance::CalculateStdOfNeighboringPixels(uchar *O, int i, int hKsize,
                                               int nCols, int noNeighboursPix,
                                               float mean) {
  uint32_t sum_dev = 0.0;
```

```cpp
  float Std = 0.0;
  sum_dev = 0.0;
  Std = 0.0;
  for (int j = -hKsize; j < hKsize; j++) {
    for (int k = -hKsize; k < hKsize; k++) {
      // sum_dev += pow((O[i + j * nCols + k] - mean), 2);
      sum_dev += SoilMath::quick_pow2((O[i + j * nCols + k] - mean));
    }
  }
  // Std = sqrt(sum_dev / noNeighboursPix);
  Std = SoilMath::fastPow((static_cast<double>(sum_dev) / noNeighboursPix), 2);
  return Std;
}

/*! Calculate the sum of the neighboring pixels
\param O uchar pointer to the current pixel of the original image
\param i current counter
\param hKsize half the kernelsize
\param nCols total number of columns
\param sum Total sum of the neighboringpixels
*/
void Enhance::CalculateSumOfNeighboringPixels(uchar *O, int i, int hKsize,
                                              int nCols, uint32_t &sum) {
  for (int j = -hKsize; j < hKsize; j++) {
    for (int k = -hKsize; k < hKsize; k++) {
      sum += O[i + j * nCols + k];
    }
  }
}

/*! Homebrew AdaptiveContrastStretch function which calculate the mean and
standard deviation from the neighboring pixels if the current pixel is higher
then the mean the value is incremented with an given factor multiplied with the
standard deviation, and decreased if it’s lower then the mean.
\param src cv::Mat source image
\param dst cv::Mat destination image
\param kernelsize an uchar which represent the kernelsize should be an uneven
number higher than two
\param factor float which indicates the amount the effect should take place
standard value is 1.0 only used in the adaptive contrast stretch enhancement
*/
void Enhance::AdaptiveContrastStretch(const Mat &src, Mat &dst,
                                      uchar kernelsize, float factor) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  AdaptiveContrastStretch(kernelsize, factor);
  dst = ProcessedImg;
}

/*! Homebrew AdaptiveContrastStretch function which calculate the mean and
standard deviation from the neighboring pixels if the current pixel is higher
then the mean the value is incremented with an given factor multiplied with the
standard deviation, and decreased if it’s lower then the mean.
\param kernelsize an uchar which represent the kernelsize should be an uneven
number higher than two
\param factor float which indicates the amount the effect should take place
standard value is 1.0 only used in the adaptive contrast stretch enhancement
\param chain use the results from the previous operation default value = false;
*/
void Enhance::AdaptiveContrastStretch(uchar kernelsize, float factor,
                                      bool chain) {
  // Exception handling
  EMPTY_CHECK(OriginalImg);
  if (kernelsize < 3 || (kernelsize % 2) == 0) {
    throw Exception::WrongKernelSizeException();
  }
  CV_Assert(OriginalImg.depth() != sizeof(uchar));

  // Make the pointers to the Data
  uchar *O;
  CHAIN_PROCESS(chain, O, uchar);
  uchar *P = ProcessedImg.data;

  int i = 0;
  int hKsize = kernelsize / 2;
  int nCols = OriginalImg.cols;
  int pStart = (hKsize * nCols) + hKsize + 1;

  int nData = OriginalImg.rows * OriginalImg.cols;
  int pEnd = nData - pStart;
  uint32_t noNeighboursPix = kernelsize * kernelsize;
  uint32_t sum;
  float mean = 0.0;
```

```

  uchar *nRow = GetNRow(nData, hKsize, nCols, OriginalImg.rows);

  i = pStart;
  while (i++ < pEnd) {
    // Checks if pixel isn’t a border pixel and progresses to the new row
    if (nRow[i] == 1) {
      i += kernelsize;
    }

    // Fill the neighboring pixel array
    sum = 0;
    mean = 0;

    // Calculate the statistics
    CalculateSumOfNeighboringPixels(O, i, hKsize, nCols, sum);
    mean = (float)(sum / noNeighboursPix);
    float Std = CalculateStdOfNeighboringPixels(O, i, hKsize, nCols,
                                                 noNeighboursPix, mean);

    // Stretch

    if (O[i] > mean) {
      // int addValue = O[i] + (int)(round(factor * Std));
      int addValue = O[i] + static_cast<int>(round(factor * Std));
      if (addValue < 255) {
        P[i] = addValue;
      } else {
        P[i] = 255;
      }
    } else if (O[i] < mean) {
      // int subValue = O[i] - (int)(round(factor * Std));
      int subValue = O[i] - static_cast<int>(round(factor * Std));
      if (subValue > 0) {
        P[i] = subValue;
      } else {
        P[i] = 0;
      }
    } else {
      P[i] = O[i];
    }
  }

  // Stretch the image with an normal histogram equalization
  HistogramEqualization(true);

  delete[] nRow;
}

/*! Blurs the image with a NxN kernel
\param src cv::Mat source image
\param dst cv::Mat destination image
\param kernelsize an uchar which represent the kernelsize should be an uneven
number higher than two
*/
void Enhance::Blur(const Mat &src, Mat &dst, uchar kernelsize) {
  OriginalImg = src;
  ProcessedImg.create(OriginalImg.size(), CV_8UC1);
  Blur(kernelsize);
  dst = ProcessedImg;
}

/*! Blurs the image with a NxN kernel
\param kernelsize an uchar which represent the kernelsize should be an uneven
number higher than two
\param chain use the results from the previous operation default value = false;
*/
void Enhance::Blur(uchar kernelsize, bool chain) {
  // Exception handling
  EMPTY_CHECK(OriginalImg);
  if (kernelsize < 3 || (kernelsize % 2) == 0) {
    throw Exception::WrongKernelSizeException();
  }
  CV_Assert(OriginalImg.depth() != sizeof(uchar));

  // Make the pointers to the Data
  uchar *O;
  CHAIN_PROCESS(chain, O, uchar);
  uchar *P = ProcessedImg.data;

  int nData = OriginalImg.rows * OriginalImg.cols;
  int hKsize = kernelsize / 2;
  int nCols = OriginalImg.cols;
```

```
  int pStart = (hKsize * nCols) + hKsize + 1;
  int pEnd = nData - pStart;
  int noNeighboursPix = kernelsize * kernelsize;
  uint32_t sum;

  uint32_t i;
  uchar *nRow = GetNRow(nData, hKsize, nCols, OriginalImg.rows);
  i = pStart;
  while (i++ < pEnd) {
    // Checks if pixel isn't a border pixel and progresses to the new row
    if (nRow[i] == 1) {
      i += kernelsize;
    }

    // Calculate the sum of the kernel
    sum = 0;
    CalculateSumOfNeighboringPixels(O, i, hKsize, nCols, sum);

    P[i] = (uchar)(round(sum / noNeighboursPix));
  }

  delete[] nRow;
}

/*! Stretches the image using a histogram
\param chain use the results from the previous operation default value = false;
*/
void Enhance::HistogramEqualization(bool chain) {
  // Exception handling
  EMPTY_CHECK(OriginalImg);
  CV_Assert(OriginalImg.depth() != sizeof(uchar));

  // Make the pointers to the Data
  uchar *O;
  CHAIN_PROCESS(chain, O, uchar);
  uchar *P = ProcessedImg.data;

  // Calculate the statics of the whole image
  ucharStat_t imgStats(O, OriginalImg.rows, OriginalImg.cols);
  float sFact;
  if (imgStats.min != imgStats.max) {
    sFact = 255.0f / (imgStats.max - imgStats.min);
  } else {
    sFact = 1.0f;
  }

  uint32_t i = 256;
  uchar LUT_changeValue[256];
  while (i-- > 0) {
    LUT_changeValue[i] = (uchar)(((float)(i)*sFact) + 0.5f);
  }

  O = OriginalImg.data;

  i = OriginalImg.cols * OriginalImg.rows + 1;
  while (i-- > 0) {
    *P++ = LUT_changeValue[*O++ - imgStats.min];
  }
}
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
#define ENHANCE_VERSION 1

#include "ImageProcessing.h"
#include "../SoilMath/SoilMath.h"

using namespace std;
using namespace SoilMath;

namespace Vision {
class Enhance : public ImageProcessing {
private:
  void CalculateSumOfNeighboringPixels(uchar *O, int i, int hKsize, int nCols,
                                       uint32_t &sum);
  float CalculateStdOfNeighboringPixels(uchar *O, int i, int hKsize, int nCols,
                                        int noNeighboursPix, float mean);

public:
  /*! Enumerator indicating the requested enhancement operation*/
  enum EnhanceOperation {
    _AdaptiveContrastStretch, /*!< custom adaptive contrast stretch operation*/
    _Blur,                    /*!< Blur operation*/
    _HistogramEqualization    /*!< Histogram equalization*/
  };

  Enhance();
  Enhance(const Mat &src);
  Enhance(const Mat &src, Mat &dst, uint8_t kernelsize = 9, float factor = 1.0,
          EnhanceOperation operation = _Blur);
  Enhance(const Enhance &rhs);

  ~Enhance();

  Enhance &operator=(Enhance rhs);

  void AdaptiveContrastStretch(uint8_t kernelsize, float factor,
                               bool chain = false);
  void AdaptiveContrastStretch(const Mat &src, Mat &dst, uint8_t kernelsize,
                               float factor);

  void Blur(uint8_t kernelsize, bool chain = false);
  void Blur(const Mat &src, Mat &dst, uint8_t kernelsize);

  void HistogramEqualization(bool chain = false);
  void HistogramEqualization(const Mat &src, Mat &dst);
};
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

/*! \class ImageProcessing
\brief Core class of all the image classes
Core class of all the image classes with a few commonly shared functions and
variables
*/
#include "ImageProcessing.h"

namespace Vision {
/*! Constructor of the core class*/
ImageProcessing::ImageProcessing() {}

/*! De-constructor of the core class*/
ImageProcessing::~ImageProcessing() {}

/*! Create a LUT indicating which iteration variable i is the end of an row
\param nData an int indicating total pixels
\param hKsize int half the size of the kernel, if any. which acts as an offset
from the border pixels
\param nCols int number of columns in a row
\return array of uchars where a zero is a middle column and a 1 indicates an end
of an row minus the offset from half the kernel size
*/
uchar *ImageProcessing::GetNRow(int nData, int hKsize, int nCols,
                                uint32_t totalRows) {
  // Create LUT to determine when there is an new row
  uchar *nRow = new uchar[nData + 1]{};
  // int i = 0;
  int shift = nCols - hKsize - 1;
  for (uint32_t i = 0; i < totalRows; i++) {
    nRow[(i * nCols) + shift] = 1;
  }
  return nRow;
}

std::vector<Mat> ImageProcessing::extractChannel(const Mat &src) {
  vector<Mat> chans;
  split(src, chans);
  return chans;
}

boost::signals2::connection
ImageProcessing::connect_Progress(const Progress_t::slot_type &subscriber) {
  return prog_sig.connect(subscriber);
}
}
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
#include "ImageProcessing.h"
#include "ConversionNotSupportedException.h"

namespace Vision {
class Conversion : public ImageProcessing {
public:
  /*! Enumerator which indicates the colorspace used*/
  enum ColorSpace {
    CIE_lab,   /*!< CIE La*b* colorspace */
    CIE_XYZ,   /*!< CIE XYZ colorspace */
    RI,        /*!< Redness Index colorspace */
    RGB,       /*!< RGB colorspace */
    Intensity, /*!< Grayscale colorspace */
    None       /*!< none */
  };
  ColorSpace OriginalColorSpace;  /*!< The original colorspace*/
  ColorSpace ProcessedColorSpace; /*!< The destination colorspace*/

  Conversion();
  Conversion(const Mat &src);
  Conversion(const Conversion &rhs);

  ~Conversion();

  Conversion &operator=(Conversion rhs);

  void Convert(ColorSpace convertFrom, ColorSpace convertTo,
               bool chain = false);
  void Convert(const Mat &src, Mat &dst, ColorSpace convertFrom,
               ColorSpace convertTo, bool chain = false);

private:
  /*!< Conversion matrix used in the conversion between RGB and CIE XYZ*/
  float XYZmat[3][3] = {{0.412453, 0.357580, 0.180423},
                        {0.212671, 0.715160, 0.072169},
                        {0.019334, 0.119194, 0.950227}};

  float whitePoint[3] = {
      0.9504, 1.0000, 1.0889}; /*!< Natural whitepoint in XYZ colorspace D65
                                  according to Matlab */
  // float whitePoint[3] = { 0.9642, 1.0000, 0.8251 }; /*!< Natural whitepoint
  // in XYZ colorspace D50 according to Matlab */

  void Lab2RI(float *O, float *P, int nData);
  void RGB2XYZ(uchar *O, float *P, int nData);
  void XYZ2Lab(float *O, float *P, int nData);
  void RGB2Intensity(uchar *O, uchar *P, int nData);
  inline float f_xyz2lab(float t);
};
}
```