```cpp
/*! \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);
    }

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

    /*! 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;
```

```
42
43          uchar Rstd = 0;
44          uchar Lstd = 0;
45          uchar Rmean = 0;
46          uchar Lmean = 0;
47
48          // Iterate till optimum Threshold is found between back- & foreground
49          while (T.first != T.second)
50          {
51              // Gets an array of the left part of the histogram
52              uint32_t i = T.first;
53              uint32_t *Left = new uint32_t[i] { };
54              while (i-- > 0) { Left[i] = OriginalImgStats.bins[i]; }
55
56              // Gets an array of the right part of the histogram
57              uint32_t rightEnd = 256 - T.first;
58              uint32_t *Right = new uint32_t[rightEnd] { };
59              i = rightEnd;
60              while (i-- > 0) { Right[i] = OriginalImgStats.bins[i + T.first]; }
61
62              // Calculate the statistics of both histograms,
63              // taking into account the current threshold
64              ucharStat_t sLeft(Left, 0, T.first);
65              ucharStat_t sRight(Right, T.first, 256);
66
67              // Calculate the new threshold the mean of the means
68              T.second = T.first;
69              T.first = (uchar)(((sLeft.Mean + sRight.Mean) / 2) + 0.5);
70
71              Rmean = (uchar)(sRight.Mean + 0.5);
72              Lmean = (uchar)(sLeft.Mean + 0.5);
73              Rstd = (uchar)(sRight.Std + 0.5);
74              Lstd = (uchar)(sLeft.Std + 0.5);
75          }
76
77          // Assumes the pixel value of the sought object lies between 2 sigma
78          switch (TypeObject)
79          {
80          case Bright:
81              T.first = Rmean - (3 * Rstd);
82              break;
```

```
83          case Dark:
84              T.first = Lmean + (3 * Lstd);
85              break;
86          }
87
88          return T.first;
89      }
90
91      /*! Convert a greyscale image to a BW using an automatic Threshold
92      \param src is the source image as a cv::Mat
93      \param dst destination image as a cv::Mat
94      \param TypeObject is an enumerator indicating if the bright or the dark pixels are the object and should be set to one  */
95      void Segment::ConvertToBW(const Mat &src, Mat &dst, TypeOfObjects Typeobjects)
96      {
97          OriginalImg = src;
98          ProcessedImg.create(OriginalImg.size(), CV_8UC1);
99          LabelledImg.create(OriginalImg.size(), CV_16UC1);
100         ConvertToBW(Typeobjects);
101         dst = ProcessedImg;
102     }
103
104     /*! Convert a greyscale image to a BW using an automatic Threshold
105     \param TypeObject is an enumerator indicating if the bright or the dark pixels are the object and should be set to one  */
106     void Segment::ConvertToBW(TypeOfObjects Typeobjects)
107     {
108         // Determine the threshold
109         uchar T = GetThresholdLevel(Typeobjects);
110
111         // Threshold the picture
112         Threshold(T, Typeobjects);
113     }
114
115     /*! Convert a greyscale image to a BW
116     \param t uchar set the value which is the tiping point
117     \param TypeObject is an enumerator indicating if the bright or the dark pixels are the object and should be set to one  */
118     void Segment::Threshold(uchar t, TypeOfObjects Typeobjects)
119     {
120         // Exception handling
121         EMPTY_CHECK(OriginalImg);
122         CV_Assert(OriginalImg.depth() != sizeof(uchar) ||
123                     OriginalImg.depth() != sizeof(uint16_t));
```

```
124
125         uint32_t i = 0;
126
127         // Create LUT
128         uchar LUT_newValue[256] { 0 };
129         if (Typeobjects == Bright)
130         {
131             i = 256;
132             while (i-- > t) { LUT_newValue[i] = 1; }
133         }
134         else
135         {
136             i = t + 1;
137             while (i-- > 0) { LUT_newValue[i] = 1; }
138         }
139
140         // Create the pointers to the data
141         uchar *P = ProcessedImg.data;
142         uchar *O = OriginalImg.data;
143
144         // Fills the ProcessedImg with either a 0 or 1
145         i = OriginalImg.cols * OriginalImg.rows + 1;
146         while (i-- > 0) { *P++ = LUT_newValue[*O++]; }
147     }
148
149     /*! Set all the border pixels to a set value
150     \param *P uchar pointer to the Mat.data
151     \param setValue uchar the value which is written to the border pixels
152     */
153     void Segment::SetBorder(uchar *P, uchar setValue)
154     {
155         // Exception handling
156         EMPTY_CHECK(OriginalImg);
157         CV_Assert(OriginalImg.depth() != sizeof(uchar) ||
158                   OriginalImg.depth() != sizeof(uint16_t));
159
160         uint32_t nData = OriginalImg.cols * OriginalImg.rows;
161
162         // Set borderPixels to 2
163         uint32_t i = 0;
164         uint32_t pEnd = OriginalImg.cols + 1;
165
```

```
166         // Set the top row to value 2
167         while (i < pEnd) { P[i++] = setValue; }
168
169         // Set the bottom row to value 2
170         i = nData + 1;
171         pEnd = nData - OriginalImg.cols;
172         while (i-- > pEnd) { P[i] = setValue; }
173
174         //Sets the first and the last Column to 2
175         i = 1;
176         pEnd = OriginalImg.rows;
177         while (i < pEnd)
178         {
179             P[(i * OriginalImg.cols) - 1] = setValue;
180             P[(i++ * OriginalImg.cols)] = setValue;
181         }
182     }
183
184     /*! Remove the blobs that are connected to the border
185     \param conn set the pixel connection eight or four
186     \param chain use the results from the previous operation default value = false;
187     */
188     void Segment::RemoveBorderBlobs(bool chain /*= false*/, Connected conn /*= Eight*/)
189 {
190         // Exception handling
191         CV_Assert(OriginalImg.depth() != sizeof(uchar));
192         EMPTY_CHECK(OriginalImg);
193
194         // make Pointers
195         uchar *O;
196         CHAIN_PROCESS(chain, O, uchar);
197         uchar *P = ProcessedImg.data;
198
199         // Set the border of the processed image to 2
200         //SetBorder(P, 2);
201         uint32_t nData = OriginalImg.cols * OriginalImg.rows;
202
203         // Set borderPixels to 2
204         uint32_t i = 0;
205         uint32_t pEnd = OriginalImg.cols + 1;
206
```

```
207         // Set the top row to value 2
208         while (i < pEnd) { P[i++] = 2; }
209
210         // Set the bottom row to value 2
211         i = nData + 1;
212         pEnd = nData - OriginalImg.cols;
213         while (i-- > pEnd) { P[i] = 2; }
214
215         //Sets the first and the last Column to 2
216         i = 1;
217         pEnd = OriginalImg.rows;
218         while (i < pEnd)
219         {
220             P[(i * OriginalImg.cols) - 1] = 2;
221             P[(i++ * OriginalImg.cols)] = 2;
222         }
223
224         // Iterates through the data and sets all border connected Blobs to 2;
225         uint32_t nCols = OriginalImg.cols;
226         uint32_t nRows = OriginalImg.rows;
227         nData = nCols * nRows;
228         i = OriginalImg.cols + 2;
229         pEnd = nData - OriginalImg.cols;
230
231         if (conn == Four)
232         {
233             while (i < pEnd)
234             {
235                 if (O[i] == 1 && P[i] != 2)
236                 {
237                     if (P[i - 1] == 2 || P[i - nCols] == 2) { P[i] = 2; }
238                     else { P[i] = 1; }
239                 }
240                 else if (O[i] == 0){ P[i] = 0; }
241                 else if (O[i] > 1 || O[i] < 0){ throw Exception::PixelValueOutOfBoundException(); }
242                 i++;
243             }
244         }
245         else
246         {
247             while (i < pEnd)
248             {
```

```
249                 if (O[i] == 1 && P[i] != 2)
250                 {
251                     if (P[i - 1] == 2 ||
252                         P[i - nCols] == 2 ||
253                         P[i - nCols - 1] == 2 ||
254                         P[i - nCols + 1] == 2)
255                     {
256                         P[i] = 2;
257                     }
258                     else { P[i] = 1; }
259                 }
260                 else if (O[i] == 0){ P[i] = 0; }
261                 else if (O[i] > 1 || O[i] < 0) { throw Exception::PixelValueOutOfBoundException(); }
262                 i++;
263             }
264 
265         }
266 
267         // Change values 2 -> 0
268         uchar LUT_newValue[3] { 0, 1, 0 };
269 
270         // P = ProcessedImg.data;
271 
272         i = 0;
273         pEnd = nData + 1;
274         while (i < pEnd)
275         {
276             P[i] = LUT_newValue[P[i]];
277             i++;
278         }
279     }
280 
281     /*! Label all the individual blobs in a BW source image. The result are written to the labelledImg as an ushort
282     \param conn set the pixel connection eight or four
283     \param chain use the results from the previous operation default value = false;
284     \param minBlobArea minimum area when an artifact is considered a blob
285     */
286     void Segment::LabelBlobs(bool chain, uint16_t minBlobArea, Connected conn)
287 {
288         // Exception handling
289         CV_Assert(OriginalImg.depth() != sizeof(uchar));
290         EMPTY_CHECK(OriginalImg);
```

```
291
292         // make the Pointers
293         uchar *O;
294         CHAIN_PROCESS(chain, O, uchar);
295         uint16_t *P = (uint16_t *)LabelledImg.data;
296
297         uint32_t nCols = OriginalImg.cols;
298         uint32_t nRows = OriginalImg.rows;
299         uint32_t nData = nCols * nRows;
300         uint32_t i = OriginalImg.cols + 2;
301         uint32_t j = 4;
302         uint32_t pEnd = nData - OriginalImg.cols;
303
304         uint16_t currentlbl = 0;
305         vector<vector<uint16_t>> connectedLabels;
306         vector<uint16_t> zeroVector;
307         zeroVector.push_back(currentlbl);
308         connectedLabels.push_back(zeroVector);
309
310         /* Four connected strategy... Although it's more code. If I place this check here it's less machine instructions compared to ↵
              doing it's done
311         inside the loop */
312         if (conn == Four)
313         {
314             // Loop through the picture
315             while (i < pEnd)
316             {
317                 // If current value = zero processed value = zero
318                 if (O[i] == 0) { P[i] = 0; }
319
320                 // If current value = 1 check North and West and act accordingly
321                 else if (O[i] = 1)
322                 {
323                     uint16_t North = P[i - nCols];
324                     uint16_t West = P[i - 1];
325                     uint16_t minVal;
326                     uint16_t maxVal;
327
328                     // If North and West are both zero assume this is a new blob
329                     if (North == 0 && West == 0)
330                     {
```

```
331                     P[i] = ++currentlbl;
332                     vector<uint16_t> cVector;
333                     cVector.push_back(currentlbl);
334                     connectedLabels.push_back(cVector);
335                 }
336
337                 //Sets the processed value to the smallest non - zero value of North and West and update the connectedLabels
338                 else
339                 {
340                     maxVal = SoilMath::Max(North, West);
341                     if (North == 0 || West == 0) { minVal = maxVal; }
342                     else { minVal = SoilMath::Min(North, West); }
343
344                     P[i] = minVal;
345
346                     /* If North and West belong to two different connected components set the current processed value to the
                          lowest value and remember that the highest value should be the lowest value */
347                     if (North != 0 && West != 0 && maxVal != minVal) { connectedLabels[maxVal].push_back(minVal); }
348
349                 }
350             }
351
352             // If there is a value greater then 1 or smaller then 1 throw error
353             else { throw Exception::PixelValueOutOfBoundException(); }
354             i++;
355         }
356     }
357
358     // If eight connected is required
359     else
360     {
361         // Loop through the picture
362         while (i < pEnd)
363         {
364             // If current value = zero processed value = zero
365             if (O[i] == 0) { P[i] = 0; }
366
367             // If current value = 1 check North and West and act accordingly
368             else if (O[i] = 1)
369             {
370                 uint16_t *nPixels = new uint16_t[4];
371                 nPixels[0] = P[i - 1];
```

```
372                 nPixels[1] = P[i - nCols - 1];
373                 nPixels[2] = P[i - nCols];
374                 nPixels[3] = P[i - nCols + 1];
375                 uint16_t minVal;
376                 uint16_t maxVal;
377
378                 // Sort the neighbors for easier checking
379                 SoilMath::Sort::QuickSort<uint16_t>(nPixels, 4);
380
381                  //If North NorthWest, NorthEast and West are all zero assume this is a new blob
382                 if (nPixels[3] == 0)
383                 {
384                     P[i] = ++currentlbl;
385                     vector<uint16_t> cVector;
386                     cVector.push_back(currentlbl);
387                     connectedLabels.push_back(cVector);
388                 }
389
390                 // Sets the processed value to the smallest non-zero value of North and West and update the connectedLabels
391                 else
392                 {
393                     maxVal = nPixels[3];
394
395                     // If there is only 1 neighbor of importance
396                     if (nPixels[2] == 0) { minVal = nPixels[3]; }
397                     else if (nPixels[1] == 0) { minVal = nPixels[2]; }
398                     else if (nPixels[0] == 0) { minVal = nPixels[2]; }
399                     else { minVal = nPixels[0]; }
400
401                     P[i] = minVal;
402
403                      /* If North NorthWest, NorthEast and West belong to different connected components set the current processed ↵
                            value to the lowest value and remember that the other value should be the lowest value*/
404                     if (nPixels[0] != nPixels[3])
405                     {
406                         j = 4;
407                         while (j-- > 0)
408                         {
409                             if (nPixels[j] != 0 && nPixels[j] > minVal) { connectedLabels[nPixels[j]].push_back(minVal);    }
410                         }
411                     }
412                 }
```

```
413                 }
414                 // If there is a value greater then 1 or smaller then 1 throw error
415                 else { throw Exception::PixelValueOutOfBoundException(); }
416                 i++;
417             }
418         }
419
420         // Sort all the vectors so the min value is easily obtained
421         i = currentlbl + 1;
422         while (i-- > 0) { std::sort(connectedLabels[i].begin(), connectedLabels[i].end()); }
423
424         // Create the LUT
425         uint16_t *LUT_newVal = new uint16_t[currentlbl + 1];
426         i = currentlbl + 1;
427         while (i-- > 0)
428         {
429             // If the value has a chain, crawl in that rabbit hole till the
430             // lowest value is found and sets the LUT
431             if (connectedLabels[i].size() > 1)
432             {
433
434                 uint16_t pChainVal = connectedLabels[connectedLabels[i][0]][0];
435                 uint16_t cChainVal = connectedLabels[i][0];
436                 uint16_t lowestVal = pChainVal;
437
438                 // How far goes the rabbit hole
439                 while (pChainVal != cChainVal)
440                 {
441                     cChainVal = connectedLabels[pChainVal][0];
442                     pChainVal = connectedLabels[cChainVal][0];
443                     lowestVal = pChainVal;
444                 }
445
446                 // Write the lowest label to the Look-Up-Table
447                 LUT_newVal[i] = lowestVal;
448             }
449             else { LUT_newVal[i] = i; } // End of the line so use the same label
450
451         }
452
453         // Make the labels consecutive numbers
454         uint16_t *tempLUT = new uint16_t[currentlbl + 1];
```

```
455         makeConsecutive(currentlbl, tempLUT, LUT_newVal);
456
457         // Get the maximum value
458         i = 0;
459         while (i <= currentlbl)
460         {
461             if (LUT_newVal[i] > MaxLabel) { MaxLabel = LUT_newVal[i]; }
462             i++;
463         }
464
465         // Second loop through each pixel to replace them with corresponding intermediate value
466         i = 0;
467         while (i < pEnd)
468         {
469             P[i] = LUT_newVal[P[i]];
470             i++;
471         }
472
473         // Create a LUT_filter for each value that is smaller then minBlobArea
474         SoilMath::Stats<uint16_t, uint32_t, uint64_t> ProcImgStats(P, nCols, nRows, MaxLabel, 0, MaxLabel);
475         LUT_newVal = new uint16_t[MaxLabel + 1] { };
476         uint16_t count = 0;
477         i = 0;
478         while (i <= MaxLabel)
479         {
480             if (ProcImgStats.bins[i] > minBlobArea) { LUT_newVal[i] = count++; }
481             i++;
482         }
483
484         noOfFilteredBlobs = MaxLabel - count - 1;
485         MaxLabel = count - 1;
486
487         // third loop through each pixel to replace them with corresponding final value
488         i = 0;
489         while (i < pEnd)
490         {
491             P[i] = LUT_newVal[P[i]];
492             i++;
493         }
494     }
495
496     /*! Create a BW image with only edges from a BW image
```

```
497     \param src source image as a const cv::Mat
498     \param dst destination image as a cv::Mat
499     \param conn set the pixel connection eight or four
500     \param chain use the results from the previous operation default value = false;
501     */
502     void Segment::GetEdges(const Mat &src, Mat &dst, bool chain, Connected conn)
503     {
504         OriginalImg = src;
505         GetEdges(chain, conn);
506         dst = ProcessedImg;
507     }
508
509     /*! Create a BW image with only edges from a BW image
510     \param conn set the pixel connection eight or four
511     \param chain use the results from the previous operation default value = false;
512     */
513     void Segment::GetEdges(bool chain, Connected conn)
514     {
515         // Exception handling
516         CV_Assert(OriginalImg.depth() != sizeof(uchar));
517         EMPTY_CHECK(OriginalImg);
518
519         // make Pointers
520         uchar *O;
521         CHAIN_PROCESS(chain, O, uchar);
522         uchar *P = ProcessedImg.data;
523
524         uint32_t nCols = OriginalImg.cols;
525         uint32_t nRows = OriginalImg.rows;
526         uint32_t nData = nCols * nRows;
527         uint32_t pEnd = nData + 1;
528         uint32_t i = 0;
529
530         //Loop through the image and set each pixel which has a zero neighbor set it to two.
531         if (conn = Four)
532         {
533             // Loop through the picture
534             while (i < pEnd)
535             {
536                 // If current value = zero processed value = zero
537                 if (O[i] == 0) { P[i] = 0; }
```

```
538                 // If current value = 1 check North West, South and East and act accordingly
539                 else if (O[i] = 1)
540                 {
541                     uchar *nPixels = new uchar[4];
542                     nPixels[0] = O[i - 1];
543                     nPixels[1] = O[i - nCols];
544                     nPixels[2] = O[i + 1];
545                     nPixels[3] = O[i + nCols];
546 
547                     // Sort the neighbors for easier checking
548                     SoilMath::Sort::QuickSort<uchar>(nPixels, 4);
549                     if (nPixels[0] == 0) { P[i] = 1; }
550                     else { P[i] = 0; }
551                 }
552                 else { throw Exception::PixelValueOutOfBoundException(); }
553                 i++;
554             }
555         }
556         else
557         {
558             // Loop through the picture
559             while (i < pEnd)
560             {
561                 // If current value = zero processed value = zero
562                 if (O[i] == 0) { P[i] = 0; }
563                 // If current value = 1 check North West, South and East and act accordingly
564                 else if (O[i] = 1)
565                 {
566                     uchar *nPixels = new uchar[8];
567                     nPixels[0] = O[i - 1];
568                     nPixels[1] = O[i - nCols];
569                     nPixels[2] = O[i - nCols - 1];
570                     nPixels[3] = O[i - nCols + 1];
571                     nPixels[4] = O[i + 1];
572                     nPixels[5] = O[i + nCols + 1];
573                     nPixels[6] = O[i + nCols];
574                     nPixels[7] = O[i + nCols - 1];
575 
576                     // Sort the neighbors for easier checking
577                     SoilMath::Sort::QuickSort<uchar>(nPixels, 8);
578 
```

```
579                 if (nPixels[0] == 0) { P[i] = 1; }
580                 else { P[i] = 0; }
581             }
582             else { throw Exception::PixelValueOutOfBoundException(); }
583             i++;
584         }
585     }
586 }
587
588 void Segment::GetEdgesEroding(bool chain)
589 {
590     // Exception handling
591     CV_Assert(OriginalImg.depth() != sizeof(uchar));
592     EMPTY_CHECK(OriginalImg);
593
594     // make Pointers
595     uchar *O;
596     CHAIN_PROCESS(chain, O, uchar);
597     uchar *P = ProcessedImg.data;
598
599     uint32_t nCols = OriginalImg.cols;
600     uint32_t nRows = OriginalImg.rows;
601     uint32_t nData = nCols * nRows;
602     uint32_t pEnd = nData + 1;
603     uint32_t i = 0;
604
605     // Setup the erosion
606     MorphologicalFilter eroder;
607     if (chain) {    eroder.OriginalImg = TempImg; }
608     else { eroder.OriginalImg = OriginalImg; }
609     // Setup the processed image of the eroder
610     eroder.ProcessedImg.create(OriginalImg.size(), CV_8UC1);
611     eroder.ProcessedImg.setTo(0);
612     // Setup the mask
613     Mat mask(3, 3, CV_8UC1, 1);
614     // Erode the image
615     eroder.Erosion(mask, false);
616
617     // Loop through the image and set the not eroded pixels to zero
618     while (i < pEnd)
619     {
```

```
620             if (O[i] != eroder.ProcessedImg.data[i]) { P[i] = 1; }
621             else { P[i] = 0; }
622             i++;
623         }
624         eroder.~MorphologicalFilter();
625     }
626
627     /*! Create a BlobList subtracting each individual blob out of a Labelled image. If the labelled image is empty build a new one
628       with a BW image.
629     \param conn set the pixel connection eight or four
630     \param chain use the results from the previous operation default value = false;
631     */
632     void Segment::GetBlobList(bool chain, Connected conn)
633     {
634         // Exception handling
635         CV_Assert(OriginalImg.depth() != sizeof(uchar));
636         EMPTY_CHECK(OriginalImg);
637
638         // If there isn't a labelledImg make one
639         if (MaxLabel < 1) { LabelBlobs(chain, 25, conn); }
640
641         // Make an empty BlobList
642         uint32_t i = 0;
643         uint32_t pEnd = MaxLabel + 1;
644         uint32_t nCols = OriginalImg.cols;
645         uint32_t nRows = OriginalImg.rows;
646         uint32_t nData = nCols * nRows;
647
648         Blob emptyBlob;
649         while (i < pEnd)
650         {
651             emptyBlob.Label = i;
652             emptyBlob.ROI.leftX = nCols;
653             emptyBlob.ROI.leftY = nRows;
654             emptyBlob.ROI.rightX = 0;
655             emptyBlob.ROI.rightY = 0;
656             BlobList.push_back(emptyBlob);
657             i++;
658         }
659
         // make Pointers
```

```
660         ushort *L = (ushort *)LabelledImg.data;
661
662         pEnd = nData + 1;
663         i = 0;
664         ushort currentBlob = 1;
665         uint32_t currentX, currentY;
666         uint16_t leftX, leftY, rightX, rightY, index;
667         //Loop through the labeled image and extract the Blobs
668         while (i < pEnd)
669         {
670             index = L[i];
671             if (index != 0)
672             {
673                 /* Determine the current x and y value of the current blob and
674                 sees if it is min/max */
675                 currentX = i / nCols;
676                 currentY = i % nCols;
677
678                 leftX = BlobList[index].ROI.leftX;
679                 leftY = BlobList[index].ROI.leftY;
680                 rightX = BlobList[index].ROI.rightX;
681                 rightY = BlobList[index].ROI.rightY;
682
683                 // Min value
684                 if (currentX < leftX) { BlobList[index].ROI.leftX = currentX; }
685                 if (currentY < leftY) { BlobList[index].ROI.leftY = currentY; }
686
687                 // Max value
688                 if (currentX > rightX)
689                 {
690                     BlobList[index].ROI.rightX = currentX;
691                 }
692                 if (currentY > rightY)
693                 {
694                     BlobList[index].ROI.rightY = currentY;
695                 }
696             }
697             i++;
698         }
699
700         // Loop through the BlobList and finalize it
```

```
701         i = 1;
702         pEnd = MaxLabel + 1;
703         ushort *LUT_filter = new ushort[MaxLabel + 1]{ };
704         uint32_t x, y;
705
706         while (i < pEnd)
707         {
708             LUT_filter[i] = 1;
709             // Fix swapping of x and y
710             BlobList[i].cvROI.y = BlobList[i].ROI.leftX;
711             BlobList[i].cvROI.x = BlobList[i].ROI.leftY;
712             BlobList[i].cvROI.height = BlobList[i].ROI.rightX - BlobList[i].ROI.leftX;
713             BlobList[i].cvROI.width = BlobList[i].ROI.rightY - BlobList[i].ROI.leftY;
714             if (BlobList[i].cvROI.width == 0) { BlobList[i].cvROI.width = 1; }
715             if (BlobList[i].cvROI.height == 0) { BlobList[i].cvROI.height = 1; }
716
717             BlobList[i].Img = CopyMat<ushort>(LabelledImg(BlobList[i].cvROI).clone(), LUT_filter, CV_8UC1);
718
719             LUT_filter[i] = 0;
720             i++;
721         }
722     }
723
724
725     void Segment::makeConsecutive(uint16_t LastLabelUsed, uint16_t * tempLUT, uint16_t * &LUT_newVal)
726     {
727         uint32_t i = LastLabelUsed + 1;
728         while (i-- > 0) { tempLUT[i] = LUT_newVal[i]; }
729         SoilMath::Sort::QuickSort<uint16_t>(tempLUT, LastLabelUsed + 1);
730         std::vector<uint16_t> v(LUT_newVal, LUT_newVal + (LastLabelUsed + 1));
731
732         uint16_t count = 0;
733         i = 1;
734         while (i <= LastLabelUsed)
735         {
736             if (tempLUT[i] != tempLUT[i - 1]) { std::replace(v.begin(), v.end(), tempLUT[i], ++count); }
737             i++;
738         }
739
740         LUT_newVal = &v[0];
741     }
742
```

```
743     void Segment::FillHoles(bool chain)
744     {
745         // Exception handling
746         CV_Assert(OriginalImg.depth() != sizeof(uchar));
747         EMPTY_CHECK(OriginalImg);
748 
749         // make Pointers
750         uchar *O;
751         CHAIN_PROCESS(chain, O, uchar);
752         if (chain) { ProcessedImg = TempImg.clone(); }
753         else { ProcessedImg = OriginalImg.clone(); }
754 
755         uchar *P = ProcessedImg.data;
756 
757         // Determine the starting point of the floodfill
758         int itt = -1;
759         while (P[++itt] != 0);
760         uint16_t row = static_cast<uint16_t>(itt / OriginalImg.rows);
761         uint16_t col = static_cast<uint16_t>(itt % OriginalImg.rows);
762 
763         // Fill the outside
764         //FloodFill(O, P, row, col, 2, 0);
765         cv::Rect rectangle;
766         cv::floodFill(ProcessedImg, cv::Point(col, row), cv::Scalar(2));
767 
768         // Set the unreached areas to 1 and the outside to 0;
769         uchar LUT_newVal[3] = { 1, 1, 0 };
770         uint32_t nData = OriginalImg.rows * OriginalImg.cols;
771         uint32_t i = 0;
772         while (i <= nData)
773         {
774             P[i] = LUT_newVal[P[i]];
775             i++;
776         }
777     }
778 
779     void Segment::FloodFill(uchar *O, uchar *P, uint16_t row, uint16_t col, uchar fillValue, uchar OldValue)
780     {
781         if (row < 0 || row > OriginalImg.rows) { return; }
782         if (col < 0 || col > OriginalImg.cols) { return; }
783         if (P[col + row * OriginalImg.rows] == OldValue)
```

```
784         {
785             P[col + row * OriginalImg.rows] = fillValue;
786             FloodFill(O, P, row + 1, col, fillValue, OldValue);
787             FloodFill(O, P, row, col + 1, fillValue, OldValue);
788             FloodFill(O, P, row - 1, col, fillValue, OldValue);
789             FloodFill(O, P, row, col - 1, fillValue, OldValue);
790         }
791     }
792 }
```