```cpp
/*! \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);
    }

    /*! 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();
```

```
41          break;
42      }
43      dst = ProcessedImg;
44  }
45
46  /*! Dec-constructor*/
47  Enhance::~Enhance() { }
48
49  /*! Calculate the standard deviation of the neighboring pixels
50  \param O uchar pointer to the current pixel of the original image
51  \param i current counter
52  \param hKsize half the kernelsize
53  \param nCols total number of columns
54  \param noNeighboursPix total number of neighboring pixels
55  \param mean mean value of the neighboring pixels
56  \return standard deviation
57  */
58  float Enhance::CalculateStdOfNeighboringPixels(uchar *O, int i, int hKsize, int nCols, int noNeighboursPix, float mean)
59  {
60      register float sum_dev = 0.0;
61      register float Std = 0.0;
62      int k;
63      int l;
64      sum_dev = 0.0;
65      Std = 0.0;
66      k = -hKsize;
67      while (k++ <= hKsize)
68      {
69          l = -hKsize;
70          while (l++ <= hKsize) { sum_dev += pow((O[i + k * nCols + l] - mean), 2); }
71      }
72      Std = sqrt(sum_dev / noNeighboursPix);
73      return Std;
74  }
75
76  /*! Calculate the sum of the neighboring pixels
77  \param O uchar pointer to the current pixel of the original image
78  \param i current counter
79  \param hKsize half the kernelsize
80  \param nCols total number of columns
81  \param sum Total sum of the neighboringpixels
```

```
82      */
83      void Enhance::CalculateSumOfNeighboringPixels(uchar *O, int i, int hKsize, int nCols, uint32_t &sum)
84      {
85          register int k;
86          k = -hKsize;
87          register int l;
88          while (k++ <= hKsize)
89          {
90              l = -hKsize;
91              while (l++ <= hKsize) { sum += O[i + k * nCols + l]; }
92          }
93      }
94
95      /*! 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.
96      \param src cv::Mat source image
97      \param dst cv::Mat destination image
98      \param kernelsize an uchar which represent the kernelsize should be an uneven number higher than two
99      \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
100     */
101     void Enhance::AdaptiveContrastStretch(const Mat &src, Mat &dst, uchar kernelsize, float factor)
102     {
103         OriginalImg = src;
104         ProcessedImg.create(OriginalImg.size(), CV_8UC1);
105         AdaptiveContrastStretch(kernelsize, factor);
106         dst = ProcessedImg;
107     }
108
109     /*! 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.
110     \param kernelsize an uchar which represent the kernelsize should be an uneven number higher than two
111     \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
112     \param chain use the results from the previous operation default value = false;
113     */
114     void Enhance::AdaptiveContrastStretch(uchar kernelsize, float factor, bool chain)
115     {
116         // Exception handling
```

```
117         EMPTY_CHECK(OriginalImg);
118         if (kernelsize < 3 || (kernelsize % 2) == 0) { throw Exception::WrongKernelSizeException(); }
119         CV_Assert(OriginalImg.depth() != sizeof(uchar));
120
121         // Make the pointers to the Data
122         uchar *O;
123         CHAIN_PROCESS(chain, O, uchar);
124         uchar *P = ProcessedImg.data;
125
126         register uint32_t i = 0;
127         int hKsize = kernelsize / 2;
128         int nCols = OriginalImg.cols;
129         register int pStart = (hKsize * nCols) + hKsize + 1;
130
131         int nData = OriginalImg.rows * OriginalImg.cols;
132         register int pEnd = nData - pStart;
133         uint32_t noNeighboursPix = kernelsize * kernelsize;
134         register uint32_t sum;
135         register float mean = 0.0;
136
137         uchar *nRow = GetNRow(nData, hKsize, nCols, OriginalImg.rows);
138
139         i = pStart;
140         while (i++ < pEnd)
141         {
142             // Checks if pixel isn't a border pixel and progresses to the new row
143             if (nRow[i] == 1) { i += kernelsize; }
144
145             // Fill the neighboring pixel array
146             sum = 0;
147             mean = 0;
148
149             // Calculate the statistics
150             CalculateSumOfNeighboringPixels(O, i, hKsize, nCols, sum);
151             mean = (float)(sum / noNeighboursPix);
152             float Std = CalculateStdOfNeighboringPixels(O, i, hKsize, nCols, noNeighboursPix, mean);
153
154             // Stretch
155
156             if (O[i] > mean)
157             {
```

```
158             int addValue = O[i] + (int)(round(factor * Std));
159             if (addValue < 255) { P[i] = addValue; }
160             else { P[i] = 255; }
161
162         }
163         else if (O[i] < mean)
164         {
165             int subValue = O[i] - (int)(round(factor * Std));
166             if (subValue > 0)   { P[i] = subValue; }
167             else { P[i] = 0; }
168         }
169         else { P[i] = O[i]; }
170     }
171
172     // Stretch the image with an normal histogram equalization
173     HistogramEqualization(true);
174 }
175
176 /*! Blurs the image with a NxN kernel
177 \param src cv::Mat source image
178 \param dst cv::Mat destination image
179 \param kernelsize an uchar which represent the kernelsize should be an uneven number higher than two
180 */
181 void Enhance::Blur(const Mat& src, Mat& dst, uchar kernelsize)
182 {
183     OriginalImg = src;
184     ProcessedImg.create(OriginalImg.size(), CV_8UC1);
185     Blur(kernelsize);
186     dst = ProcessedImg;
187 }
188
189 /*! Blurs the image with a NxN kernel
190 \param kernelsize an uchar which represent the kernelsize should be an uneven number higher than two
191 \param chain use the results from the previous operation default value = false;
192 */
193 void Enhance::Blur(uchar kernelsize, bool chain)
194 {
195     // Exception handling
196     EMPTY_CHECK(OriginalImg);
197     if (kernelsize < 3 || (kernelsize % 2) == 0) { throw Exception::WrongKernelSizeException(); }
198     CV_Assert(OriginalImg.depth() != sizeof(uchar));
199
```

```
200         // Make the pointers to the Data
201         uchar *O;
202         CHAIN_PROCESS(chain, O, uchar);
203         uchar *P = ProcessedImg.data;
204
205         int nData = OriginalImg.rows * OriginalImg.cols;
206         int hKsize = kernelsize / 2;
207         int nCols = OriginalImg.cols;
208         register int pStart = (hKsize * nCols) + hKsize + 1;
209         register int pEnd = nData - pStart;
210         int noNeighboursPix = kernelsize * kernelsize;
211         register uint32_t sum;
212
213         uint32_t i;
214         uchar *nRow = GetNRow(nData, hKsize, nCols, OriginalImg.rows);
215         i = pStart;
216         while (i++ < pEnd)
217         {
218             // Checks if pixel isn't a border pixel and progresses to the new row
219             if (nRow[i] == 1) { i += kernelsize; }
220
221             // Calculate the sum of the kernel
222             sum = 0;
223             CalculateSumOfNeighboringPixels(O, i, hKsize, nCols, sum);
224
225             P[i] = (uchar)(round(sum / noNeighboursPix));
226         }
227     }
228
229     /*! Stretches the image using a histogram
230     \param chain use the results from the previous operation default value = false;
231     */
232     void Enhance::HistogramEqualization(bool chain)
233     {
234         // Exception handling
235         EMPTY_CHECK(OriginalImg);
236         CV_Assert(OriginalImg.depth() != sizeof(uchar));
237
238         // Make the pointers to the Data
239         uchar *O;
240         CHAIN_PROCESS(chain, O, uchar);
```

```
241         uchar *P = ProcessedImg.data;
242
243         // Calculate the statics of the whole image
244         ucharStat_t imgStats(O, OriginalImg.rows, OriginalImg.cols);
245         register float sFact;
246         if (imgStats.min != imgStats.max) { sFact = 255.0f / (imgStats.max - imgStats.min); }
247         else { sFact = 1.0f; }
248
249         uint32_t i = 256;
250         register uchar LUT_changeValue[256];
251         while (i-- > 0) { LUT_changeValue[i] = (uchar)(((float)(i) * sFact) + 0.5f); }
252
253         O = OriginalImg.data;
254
255         i = OriginalImg.cols * OriginalImg.rows + 1;
256         while (i-- > 0) { *P++ = LUT_changeValue[*O++ - imgStats.min]; }
257     }
258 }
```