

Criterion C: Development

The libraries used in this program are as follow:

OpenCV libraries: for the built-in image processing and the image codexes (<https://opencv.org/>).

The c++ boost libraries: to easier manage the folder system (<http://www.boost.org/>).

Words 37

Complex code 1:

```

122 list<feature> CorrelateImageFeatures(list<feature>* keyFeatures)
123 {
124     list<list<pair<feature*, cv::Point>>> keywordLists;
125
126     list<feature>::iterator featureIterator = keyFeatures->begin();
127     for (int i = 0; i < keyFeatures->size(); i++)
128     {
129         list<pair<feature*, cv::Point>> keywordToBe;
130         keywordToBe.push_front(pair<feature*, cv::Point>(&*featureIterator, cv::Point(0, 0))); //added the first one
131
132         list<feature>::iterator featureIteratorFromBack = keyFeatures->end();
133         --featureIteratorFromBack; //I do this because the iterator has an item at the end that is there to say there is no more items
134         for (int j = int(keyFeatures->size()) - 1; j > i; j--)
135         {
136             float correlationThreshold;
137             GetConfigVarsFromID(ImageCorrelationThresholdTrainer) >> correlationThreshold;
138
139             pair<feature*, cv::Point> thisFeature;
140             thisFeature.first = &*featureIteratorFromBack;
141             if (DoesCorrelationReachThreshold(featureIterator->grayScale, featureIteratorFromBack->grayScale, 0.0, correlationThreshold, &thisFeature.second, false))
142             {
143                 keywordToBe.push_back(thisFeature);
144             }
145             --featureIteratorFromBack;
146         }
147         keywordLists.push_back(keywordToBe);
148
149         ++featureIterator;
150     }
151     return CreateKeywords(keywordLists, 0);
152 }
153
154
271 list<feature> CreateKeywords(list<list<pair<feature*, cv::Point>>> keywordLists, int thresholdOfSharedImages)
272 {
273     list<feature>* featuresAlreadyInKeywords;
274
275     list<feature> outPut;
276
277     list<list<pair<feature*, cv::Point>>>::iterator keywordIterator = keywordLists.begin();
278     for (int i = 0; i < keywordLists.size(); i++)
279     {
280         if (!((featuresAlreadyInKeywords.size() > 0 && find(featuresAlreadyInKeywords.begin(), featuresAlreadyInKeywords.end(), keywordIterator->begin()->first) != featuresAlreadyInKeywords.end())))
281         {
282             if (keywordIterator->size() >= thresholdOfSharedImages)
283             {
284                 feature thisFeature = *keywordIterator->begin()->first;
285
286                 cv::Point mainImageOffset(0, 0);
287
288                 list<pair<feature*, cv::Point>>::iterator keywordImageIterator = keywordIterator->begin();
289                 ++keywordImageIterator; //the first one is the header image that all the other image will be added to
290                 for (int j = 1; j < keywordIterator->size(); j++)
291                 {
292                     AddImagesAt(
293                         &thisFeature.grayScale,
294                         &(keywordImageIterator->first->grayScale),
295                         &thisFeature.grayScale,
296                         cv::Point(keywordImageIterator->second.x - mainImageOffset.x, keywordImageIterator->second.y - mainImageOffset.y),
297                         &mainImageOffset,
298                         1 - (1 / (float(j) + 1)),
299                         true);
300
301                     featuresAlreadyInKeywords.push_back(keywordImageIterator->first);
302
303                     for each (array<int, 4> range in keywordImageIterator->first->ranges)
304                         thisFeature.ranges.push_back(range);
305
306                     if (ShowImages)
307                         showImage(&(keywordImageIterator->first->grayScale));
308
309                     ++keywordImageIterator;
310                 }
311                 if (ShowImages)
312                 {
313                     showImage(&keywordIterator->begin()->first->grayScale);
314                     showImage(&thisFeature.grayScale);
315                     cv::waitKey(0);
316                 }
317
318                 outPut.push_back(thisFeature);
319             }
320         }
321         ++keywordIterator;
322     }
323     return outPut;
324 }
325
326

```

When creating the keywords, the program has to check to see if there are any duplicates of different features like a car having two wheels. It does this by creating a list of “to be” keywords that include all the keywords they are similar enough to be classified as being of the same thing. The CorralateImageFeatures method does this basic procedure. The KeywordsList list is this list (line 124). The point is the offset at which the images have the best correlation to the main image, which is at the front of the list.

Words 96

Complex code 2:

```
166 bool DoesCorrelationReachThreshold(Mat image, Mat templ, float maxRot, float threshold, Point *location, bool scaleImg2ToMatchRows)
167 {
168     try
169     {
170         Mat result(image.rows, image.cols, DataType<uchar>::type);
171
172         Concurrency::parallel_for(0, image.rows, [&](int y)//uses threading in a for loop
173         {
174             for (int x = 0; x < image.cols; x++)
175             {
176                 DoOneCorrelation(&image, &templ, maxRot, Point(x, y), &result);
177             }
178         });
179
180         double minVal, maxVal;
181         Point minLoc, maxLoc, matchLoc;
182
183         minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc, Mat());
184
185         *location = cv::Point(maxLoc.x - (templ.cols / 2), maxLoc.y - (templ.rows / 2));
186
187         if (maxVal/255 >= threshold)
188             return true;
189         else
190             return false;
191     }
192     catch (Exception ex)
193     {
194         cout << "GetGradientImage: " << ex.err << endl;
195         return false;
196     }
197 }
198
133 void DoOneCorrelation(Mat *image, Mat *templ, float maxRot, Point p, Mat *result)
134 {
135     int sharedPixels = 0;
136     double correlation = 0;
137
138     for (int templX = 0; templX < templ->cols; templX++)
139     {
140         for (int templY = 0; templY < templ->rows; templY++)
141         {
142             int imageX = p.x - (templ->cols / 2) + templX;
143             int imageY = p.y - (templ->rows / 2) + templY;
144
145             if (imageX >= 0 && imageX < image->cols && imageY >= 0 && imageY < image->rows)
146             {
147                 sharedPixels++;
148
149                 int absDiff = std::abs(templ->at<uchar>(Point(templX, templY)) - image->at<uchar>(Point(imageX, imageY)));
150                 float thisCorrelation = (1 - (float(absDiff) / 255));
151                 correlation += thisCorrelation;
152             }
153             else
154                 continue;
155         }
156     }
157
158     float sharedPixelPresent = float(sharedPixels) / (templ->cols * templ->rows);
159     correlation = ((correlation / sharedPixels) * 3 + sharedPixelPresent * 1) / 4;
160
161     uchar temp = uchar(correlation * 255);
162     result->at<uchar>(p) = temp;
163 }
164
165
```

In the previous complexed code section, there was a method that was called, called DoesCorralationReachThreshold, this method figure out whether or not two images are of roughly the same thing and their offset. To accomplish this, the program iterates through each pixel of the base image (image) and using the template image (templ) it gets the correlation of the two with the template image being offset so that the middle of the template is at the pixel mentioned above. This correlation algorithm also takes into account how similar the images are and what present overlapped and was correlated (line 160).

Words 100

Complex code 3:

```
337 int SaveOneAsImgFile(std::string folderPath, std::string folderName, feature* image)
338 {
339     boost::filesystem::path dir(folderPath + folderName);
340     if (!boost::filesystem::exists(dir))
341         if (!boost::filesystem::create_directory(dir))
342             return -1;
343
344     boost::uuids::uuid uuid = boost::uuids::random_generator()();
345     std::string filePath = folderPath + folderName + "\\" + to_string(uuid) + ".jpg";
346
347     imwrite(filePath, image->grayScale);
348
349     ofstream myfile;
350     myfile.open(folderPath + folderName + "\\" + to_string(uuid) + ".vctrinf");
351     myfile << "#Vector Info for keyword: " << uuid << endl;
352     for each (std::array<int,4> vector in image->ranges)
353     {
354         myfile << "[" << endl;
355         myfile << vector[0] << endl;
356         myfile << vector[1] << endl;
357         myfile << vector[2] << endl;
358         myfile << vector[3] << endl;
359         myfile << "]" << endl;
360     }
361     myfile.close();
362
363     return 0;
364 }
365
```

This program creates “keywords” that consists of an image and a vector pointing to the center of the object relative to the keyword’s position. This part of the code it what is used to save the keyword to the file. The start at line 339-342 the program creates the folder were the keywords will be stored with name folderName. 344 and 345 create a Guid for each keyword as its name. Then the image image->grayScale is named to the folder along with the vector information.

Words 86

Complex code 4:

```

177 list<feature> GetMostImportantPartsOfImage(
178     cv::Mat *grayImage, int maxFeatures, float xStepSizePercentOfImage,
179     float maxRot, float rotStep, float thresholdPresent, int maxfeatureSizeInSteps,
180     int minfeatureSizeInSteps)
181 {
182     using namespace cv;
183     Mat gradImg;
184     list<feature> Features;
185     gradImg = GetGradientImage(*grayImage);
186
187     int pixelsPerStep = int(gradImg.cols / (100 / xStepSizePercentOfImage)); //pps is pixels per step
188
189     for (int x = 0; x < int(ceil(gradImg.cols / double(pixelsPerStep))); x++)
190     {
191         for (int y = 0; y < int(ceil(gradImg.rows / double(pixelsPerStep))); y++)
192         {
193             for (int s = minfeatureSizeInSteps; s < maxfeatureSizeInSteps; s++)
194             {
195                 feature f;
196                 f.gradientImage = MakeMatFromRange(
197                     cv::Point(x * pixelsPerStep, y * pixelsPerStep),
198                     cv::Point((x + s) * pixelsPerStep, (y + s) * pixelsPerStep),
199                     gradImg,
200                     ShowImages);
201
202                 f.grayScale = TotalMatAddByOne(MakeMatFromRange(
203                     cv::Point(x * pixelsPerStep, y * pixelsPerStep),
204                     cv::Point((x + s) * pixelsPerStep, (y + s) * pixelsPerStep),
205                     grayImage,
206                     ShowImages));
207
208                 f.GetRating();
209                 if (f.rating < thresholdPresent)
210                     continue;
211                 std::array<int, 4> range; //for c++ you have to use std::array in order to have an array in a list
212                 range[0] = (gradImg.cols / 2) - (x) * pixelsPerStep;
213                 range[1] = ((gradImg.cols + 1) / 2) - (x - 0.5f) * pixelsPerStep;
214                 range[2] = (gradImg.rows / 2) - (y) * pixelsPerStep;
215                 range[3] = ((gradImg.rows + 1) / 2) - (y - 0.5f) * pixelsPerStep;
216                 f.ranges.push_back(range);
217
218                 Features.push_back(f);
219             }
220         }
221     }
222
223     if (Features.size() < maxFeatures / 2)
224         Features = GetMostImportantPartsOfImage(grayImage, maxFeatures, xStepSizePercentOfImage, 0, 0, 0, maxfeatureSizeInSteps, minfeatureSizeInSteps);
225
226     Features.sort(useRating);
227     while (Features.size() > maxFeatures)
228         Features.pop_back();
229
230     return Features;
231 }
232

```

When creating the keywords, the program must find potential places in each training image to be keywords. This is done by finding the places in the images with the most distinguishable details. To do this, an edge image is made using a Sobel operator of a potential keyword (line 193) defined by the 3 for loops (lines 186-190). If the potential keyword has enough detail than it is turned into a keyword having the vector pointing to the center created. In the program, the Keywords are stored in the feature class.

Words 91

Total words 433