Criterion C: Development

Theory:
This program uses multiple techniques to accomplish the end goal of having a trainable detection program. These techniques include the hough transform, image correlation, convolution, derivative kernels and many more minor but important techniques. The hough transform is used as a way to store the possible points for where the object could be. To find the possible point the keywords are correlated with convolution to find where the best match or matches are for the keyword used to correlate with. Then the vector is traced to where the predicted center is.

The program uses one main technique, keywords, these keywords are key features of an object. They consist of two major things, the feature and its relative location relating to the center of the object. The point of using keywords is having a way to train a program to detect objects. This works because keywords are common high detailed features of object that can be used to detect the object. How the training aspect works to better the ability to detect these features of the image is that the keywords are made by adding images of these features together to make something that identifies the object in a broader way. To find the places that have high detail in the image a derivative kernel, the sobel operator, it used to covolve the image. What this does is it finds the change in intensity in the for of a vector with direction and magnitude at a given point and it surroundings. Because we don't need the direction only the magnitudes are recorded in to a mat.

Words 265

libraries:
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:

```cpp
list<feature> CorrelateImageFeatures(list<feature>* keyFeatures)
{
    list<list<pair<feature*, cv::Point>>> keywordLists;

    list<feature>::iterator featureIterator = keyFeatures->begin();
    for (int i = 0; i < keyFeatures->size(); i++)
    {
        list<pair<feature*, cv::Point>> keywordToBe;
        keywordToBe.push_front(pair<feature*, cv::Point>(&*featureIterator, cv::Point(0, 0)));//added the first one

        list<feature>::iterator featureIteratorFromBack = keyFeatures->end();
        --featureIteratorFromBack;//I do this becasue the iterator has an item at the end that is there to say there is no more items
        for (int j = int(keyFeatures->size()) - 1; j > i; j--)
        {
            float corralationThreshold;
            GetConfigVarsFromID(ImageCorralationThresholdTrainer) >> corralationThreshold;

            pair<feature*, cv::Point> thisFeature;
            thisFeature.first = &*featureIteratorFromBack;
            if (DoesCorrelationReachThreshold(featureIterator->grayScale, featureIteratorFromBack->grayScale, 0.0, corralationThreshold, &thisFeature.second, false))
            {
                keywordToBe.push_back(thisFeature);
            }
            --featureIteratorFromBack;
        }
        keywordLists.push_back(keywordToBe);

        ++featureIterator;
    }
    return CreateKeywords(keywordLists, 0);

}
```

```cpp
list<feature> CreateKeywords(list<list<pair<feature*, cv::Point>>> keywordLists, int thresholdOfSharedImages)
{
    list<feature*> featuresAlreadyInKeywords;

    list<feature> outPut;

    list<list<pair<feature*, cv::Point>>>::iterator keywordIterator = keywordLists.begin();
    for (int i = 0; i < keywordLists.size(); i++)
    {
        if (!(featuresAlreadyInKeywords.size() > 0 && find(featuresAlreadyInKeywords.begin(), featuresAlreadyInKeywords.end(), keywordIterator->begin()->first) != featuresAlreadyInKeywords.end()))//
        {
            if (keywordIterator->size() >= thresholdOfSharedImages)
            {
                feature thisFeature = *keywordIterator->begin()->first;

                cv::Point mainImageOffSet(0, 0);

                list<pair<feature*, cv::Point>>::iterator keywordImageIterator = keywordIterator->begin();
                ++keywordImageIterator;//the first one is the header image that all the other image will be added to
                for (int j = 1; j < keywordIterator->size(); j++)
                {
                    AddImagesAt(
                        &thisFeature.grayScale,
                        &(keywordImageIterator->first->grayScale),
                        &thisFeature.grayScale,
                        cv::Point(keywordImageIterator->second.x - mainImageOffSet.x, keywordImageIterator->second.y - mainImageOffSet.y),
                        &mainImageOffSet,
                        1 - (1 / (float(j) + 1)),
                        true);

                    featuresAlreadyInKeywords.push_back(keywordImageIterator->first);

                    for each (array<int, 4> range in keywordImageIterator->first->ranges)
                        thisFeature.ranges.push_back(range);

                    if(ShowImages)
                        showImage(&(keywordImageIterator->first->grayScale));

                    ++keywordImageIterator;
                }
                if (ShowImages)
                {
                    showImage(&keywordIterator->begin()->first->grayScale);
                    showImage(&thisFeature.grayScale);
                    cv::waitKey(0);
                }

                outPut.push_back(thisFeature);
            }
        }
        ++keywordIterator;
    }

    return outPut;
}
```

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 thatare similar enough to be classified as being the same thing. The CorrelateImageFeatures 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:

```
bool DoesCorrelationReachThreshold(Mat image, Mat templ, float maxRot, float threshold, Point *location, bool scaleImg2ToMatchRows)
{
    try
    {
        Mat result(image.rows, image.cols, DataType<uchar>::type);

        Concurrency::parallel_for(0, image.rows, [&](int y)//uses threading in a for loop
        {
            for (int x = 0; x < image.cols; x++)
            {
                DoOneCorrelation(&image, &templ, maxRot, Point(x, y), &result);
            }
        });

        double minVal, maxVal;
        Point minLoc, maxLoc, matchLoc;

        minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc, Mat());

        *location = cv::Point(maxLoc.x - (templ.cols / 2), maxLoc.y - (templ.rows / 2));

        if (maxVal/255 >= threshold)
            return true;
        else
            return false;
    }
    catch (Exception ex)
    {
        cout << "GetGradientImage: " << ex.err << endl;
        return false;
    }
}

```

```
void DoOneCorrelation(Mat *image, Mat *templ, float maxRot, Point p, Mat *result)
{
    int sharedPixels = 0;
    double correlation = 0;

    for (int templX = 0; templX < templ->cols; templX++)
    {
        for (int templY = 0; templY < templ->rows; templY++)
        {
            int imageX = p.x - (templ->cols / 2) + templX;
            int imageY = p.y - (templ->rows / 2) + templY;

            if (imageX >= 0 && imageX < image->cols && imageY >= 0 && imageY < image->rows)
            {
                sharedPixels++;

                int absDiff = std::abs(templ->at<uchar>(Point(templX, templY)) - image->at<uchar>(Point(imageX, imageY)));
                float thisCorrelation = (1 - (float(absDiff) / 255));
                correlation += thisCorrelation;

            }
            else
                continue;
        }
    }

    float sharedPixelPresent = float(sharedPixels) / (templ->cols * templ->rows);
    correlation = ((correlation / sharedPixels) * 3 + sharedPixelPresent * 1) / 4;

    uchar temp = uchar(correlation * 255);
    result->at<uchar>(p) = temp;
}

```

In the previous complex code section, there was a method that was called DoesCorralationReachThreshold, this method figures 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:

```cpp
177 list<feature> GetMostImportantPartsOfImage(
178     cv::Mat *grayImage, int maxFeatures, float xStepSizePersentOfImage,
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 / xStepSizePersentOfImage));//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, xStepSizePersentOfImage, 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 690