MP.1 Data Buffer Optimization

Implements a ring buffer where new elements are added to tail and older are removed from head.

```
if (dataBuffer.size() > dataBufferSize) {

          // size exceeded, dequeue the oldest element

          dataBuffer.erase(dataBuffer.begin());

        }

        else {

          // enqueue the new DataFrame

          dataBuffer.push_back(frame);

        }
```

MP.2 Keypoint Detection

Implement detectors HARRIS, FAST, BRISK, ORB, AKAZE, and SIFT and make them selectable by setting a string accordingly.

1. Traditional Harris detector for keypoints detection is given by this function.

```
2. void detKeypointsHarris(std::vector<cv::KeyPoint> &keypoints,
   cv::Mat &img, bool bVis)
3. {
4.     int blockSize = 2;      // a blockSize x blockSize neighborhood
   for every pixel
```

```
    int apertureSize = 3;   // for sobel operator
    int minResponse = 100;  // minimum value for a corner in the 8-bit scaled response matrix
    double k = 0.04;        // Harris parameter

    cv::Mat dst, dst_norm, dst_norm_scaled;
    dst = cv::Mat::zeros(img.size(), CV_32FC1);

    double t = (double)cv::getTickCount();

    cv::cornerHarris(img, dst, blockSize, apertureSize, k, cv::BORDER_DEFAULT);
    cv::normalize(dst, dst_norm, 0, 255, cv::NORM_MINMAX, CV_32FC1, cv::Mat());
    cv::convertScaleAbs(dst_norm, dst_norm_scaled);

    // look for prominent corners and keypoints
    double maxOverlap = 0.0;
    for(size_t i = 0; i < dst_norm.rows; i++)
    {
        for(size_t j = 0; j < dst_norm.cols; j++)
        {
            int response = (int)dst_norm.at<float>(i,j);
            if(response > minResponse)
            {
                // only store points above a threshold
                cv::KeyPoint newKeypoint;
                newKeypoint.pt = cv::Point2f(j, i);
                newKeypoint.size = 2*apertureSize;
                newKeypoint.response = response;
                newKeypoint.class_id = 1;

                // perform non-maximal suppression in local neighbourhood around new key point
                bool bOverlap = false;
                for(auto it = keypoints.begin(); it != keypoints.end(); ++it)
                {
                    double kptOverlap = cv::KeyPoint::overlap(newKeypoint, *it);
                    if(kptOverlap > maxOverlap)
                    {
                        bOverlap = true;
                        if(newKeypoint.response > (*it).response)
                        {
                            *it = newKeypoint;
                            break;
                        }
                    }
                }

                if(!bOverlap)
                {
                    keypoints.push_back(newKeypoint);
                }
            }
        }
    }
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    std::cout << "Harris detector with n= " << keypoints.size() << " keypoints in " << 1000*t/1.0 << " ms" << std::endl;

    // visualize results
    if (bVis)
    {
        cv::Mat visImage = dst_norm_scaled.clone();
        cv::drawKeypoints(dst_norm_scaled, keypoints, visImage, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
        string windowName = "Harris Corner Detector Results";
        cv::namedWindow(windowName, 5);
        imshow(windowName, visImage);
        cv::waitKey(0);
    }
}
```

The other modern detector including FAST, BRISK, ORB, AKAZE, and SIFT are given in this function below, with parameter **detectorType**.

```
void detKeypointsModern(std::vector<cv::KeyPoint> &keypoints, cv::Mat &img,
std::string detectorType, bool bVis)
```

MP.3 Keypoint Removal

To remove all keypoints outside of a pre-defined rectangle and only use the keypoints within the rectangle for further processing.

```
  cv::Rect vehicleRect(535, 180, 180, 150);

        //std::cout << "Total keypoints: " << keypoints.size() << std::endl;

        std::vector<cv::KeyPoint> veh_kps;

        if (bFocusOnVehicle)

        {

            // Remove keypoints outside of the vehicleRect

            for (auto it=keypoints.begin(); it != keypoints.end(); it++ ) {

              if (vehicleRect.contains(it->pt)) {

                //keypoints.erase(it);

                veh_kps.push_back(*it);

              }

            }

        }
```

MP.4 Keypoint Descriptors

Implements descriptors BRIEF, ORB, FREAK, AKAZE and SIFT and make them selectable by setting a string accordingly. `string descriptorType` to select descriptor type.

```
void descKeypoints(vector<cv::KeyPoint> &keypoints, cv::Mat &img, cv::Mat
&descriptors, string descriptorType)
{
    // select appropriate descriptor
    cv::Ptr<cv::DescriptorExtractor> extractor;
    if (descriptorType.compare("BRISK") == 0)
    {
        int threshold = 30;        // FAST/AGAST detection threshold score.
        int octaves = 3;           // detection octaves (use 0 to do single
scale)
        float patternScale = 1.0f; // apply this scale to the pattern used
for sampling the neighbourhood of a keypoint.

        extractor = cv::BRISK::create(threshold, octaves, patternScale);
    }
    else if(descriptorType.compare("SIFT") == 0)
    {
        extractor = cv::xfeatures2d::SiftDescriptorExtractor::create();
    }
    else if(descriptorType.compare("ORB") == 0)
    {
        extractor = cv::ORB::create();
    }
    else if(descriptorType.compare("FREAK") == 0)
    {
        extractor = cv::xfeatures2d::FREAK::create();
    }
    else if(descriptorType.compare("AKAZE") == 0)
    {
        extractor = cv::AKAZE::create();
    }
    else if(descriptorType.compare("BRIEF") == 0)
    {
        extractor = cv::xfeatures2d::BriefDescriptorExtractor::create();
    }

    // perform feature description
    double t = (double)cv::getTickCount();
    extractor->compute(img, keypoints, descriptors);
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << descriptorType << " descriptor extraction in " << 1000 * t /
1.0 << " ms" << endl;
}
```

MP.5 Descriptor Matching && MP.6 Descriptor Distance Ratio

Implement FLANN matching as well as k-nearest neighbor selection. Both methods must be selectable using the respective strings in the main function; Use the KNN matching to implement the descriptor distance ratio test, which looks at the ratio of best vs. second-best match to decide whether to keep an associated pair of keypoints.

All these three tasks are realized in this function, k = 2; distance ratio = 0.8;

```
void matchDescriptors(std::vector<cv::KeyPoint> &kPtsSource,
std::vector<cv::KeyPoint> &kPtsRef, cv::Mat &descSource, cv::Mat &descRef,
                      std::vector<cv::DMatch> &matches, std::string
descriptorType, std::string matcherType, std::string selectorType)
{
    // configure matcher
    bool crossCheck = false;
    cv::Ptr<cv::DescriptorMatcher> matcher;

    if (matcherType.compare("MAT_BF") == 0)
    {

        int normType = descriptorType.compare("DES_BINARY") == 0 ?
cv::NORM_HAMMING : cv::NORM_L2;
        matcher = cv::BFMatcher::create(normType, crossCheck);
    }
    else if (matcherType.compare("MAT_FLANN") == 0)
    {
        if (descSource.type() != CV_32F)
        {
            // OpenCV bug workaround : convert binary descriptors to
floating point due to a bug in current OpenCV implementation
            descSource.convertTo(descSource, CV_32F);
            descRef.convertTo(descRef, CV_32F);
        }
        matcher =
cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);
    }

    // perform matching task
    if (selectorType.compare("SEL_NN") == 0)
    {
        // nearest neighbor (best match)
        double t = (double)cv::getTickCount();
        matcher->match(descSource, descRef, matches); // Finds the best
match for each descriptor in desc1
        t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
        std::cout << "NN with n=" << matches.size() << " matches in " <<
1000*t/1.0 << " ms" << std::endl;
    }
    else if (selectorType.compare("SEL_KNN") == 0)
    {
        // k nearest neighbors (k=2)
        vector<vector<cv::DMatch>> knn_matches;
        double t = (double)cv::getTickCount();
        matcher->knnMatch(descSource, descRef, knn_matches, 2);
        t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
        std::cout << "KNN with n = " << knn_matches.size() << " matches in
" << 1000*t/1.0 << " ms" << std::endl;

        // Implement k-nearest-neighbor matching and filter matches using
descriptor distance ratio test
        double minDescDistRatio = 0.8;
        for(auto it = knn_matches.begin(); it != knn_matches.end(); ++it)
        {
            if((*it)[0].distance < minDescDistRatio * (*it)[1].distance)
            {
                matches.push_back((*it)[0]);
            }
        }
        std::cout << "# keypoints removed = " << knn_matches.size() -
matches.size() << std::endl;
    }
}
```

MP.7 Keypoints Counting

To count the number of keypoints on the preceding vehicle for all 10 images and take note of the distribution of their neighborhood size. Do this for all the detectors you have implemented.

Detector	Img 0	Img 1	Img 2	Img 3	Img 4	Img 5	Img 6	Img 7	Img 8	Img 9	Average

Harris	17	14	18	21	26	43	18	31	26	34	25
Shi-Tomasi	125	118	123	120	120	113	114	123	111	112	118
FAST	149	152	150	155	149	149	156	150	138	143	149
BRISK	264	282	282	277	297	279	289	272	266	254	276
ORB	92	102	106	113	109	125	130	129	127	128	116
AKAZE	166	157	161	155	163	164	173	175	177	179	167
SIFT	138	132	124	137	134	140	137	148	159	137	138

MP.8 Matching Statistics

To count the number of matched keypoints for all 10 images using all possible combinations of detectors and descriptors. In the matching step, use the BF approach with the descriptor distance ratio set to 0.8.

Combination(detect + descriptor)	# Detected Keypoints	Detection Time	Extraction Time	#Matched Keypoint	Matching Time
Harris + SIFT	172	17.5ms	30ms	18	0.08ms
Harris + BRISK	172	17.5ms	0.94ms	16	0.33ms
Harris + ORB	172	17.5ms	2.58ms	17	0.37ms
Harris + FREAK	172	17.5ms	49.10ms	16	0.17ms
Harris + AKAZE	172	17.5ms	76.35ms	19	0.07ms
Harris + BRIEF	172	17.5ms	2.62ms	19	0.15ms
Shi-Tomasi + SIFT	1342	15.8ms	29ms	103	0.53ms
Shi-Tomasi + BRISK	1342	15.8ms	1.65ms	86	0.25ms

Shi-Tomasi + ORB	1342	15.8ms	2.68ms	101	0.91ms
Shi-Tomasi + FREAK	1342	15.8ms	50.50ms	86	0.36ms
Shi-Tomasi + AKAZE	1342	15.8ms	80.47ms	108	0.33ms
Shi-Tomasi + BRIEF	1342	15.8ms	2.55ms	110	0.74ms
FAST + SIFT	1787	1.47ms	34ms	117	0.90ms
FAST + BRISK	1787	1.47ms	1.70ms	100	0.36ms
FAST + ORB	1787	1.47ms	3.15ms	115	0.93ms
FAST + FREAK	1787	1.47ms	52ms	101	0.76ms
FAST + AKAZE	1787	1.47ms	80ms	123	0.44ms
FAST + BRIEF	1787	1.47ms	2.43ms	117	1.24ms
BRISK + SIFT	2711	40ms	51ms	180	1.68ms
BRISK + BRISK	2711	40ms	2.40ms	170	0.98ms
BRISK + ORB	2711	40ms	9.84ms	160	0.89ms
BRISK + FREAK	2711	40ms	47.84ms	161	0.86ms
BRISK + AKAZE	2711	40ms	76.50ms	156	1.50ms
BRISK + BRIEF	2711	40ms	3.2ms	186	1.67ms
ORB + SIFT	500	9ms	59.60ms	82	0.30ms
ORB + BRISK	500	9ms	1.45ms	81	0.23ms
ORB + ORB	500	9ms	11.50ms	83	0.25ms
ORB + FREAK	500	9ms	50.30ms	48	0.14ms
ORB + AKAZE	500	9ms	77.50ms	57	0.28ms
ORB + BRIEF	500	9ms	2.10ms	56	0.60ms
AKAZE + SIFT	1343	79ms	38ms	139	0.68ms
AKAZE + BRISK	1343	79ms	1.84ms	134	0.38ms
AKAZE + ORB	1343	79ms	8.80ms	130	0.55ms
AKAZE + FREAK	1343	79ms	52ms	130	0.46ms
AKAZE + AKAZE	1343	79ms	79ms	139	0.48ms
AKAZE + BRIEF	1343	79ms	2.70ms	132	1.0ms

SIFT + SIFT	1384	119ms	104ms	86	0.49ms
SIFT + BRISK	1384	119ms	1.62ms	60	0.30ms
SIFT + FREAK	1384	119ms	48.97ms	62	0.30ms
SIFT + AKAZE	1384	119ms	73ms	40	0.36ms
SIFT + BRIEF	1384	119ms	1.98ms	80	0.77ms

MP.9 Time Consumption

To log the time it takes for keypoint detection and descriptor extraction. The results must be entered into a spreadsheet and based on this information you will then suggest the TOP3 detector / descriptor combinations as the best choice for our purpose of detecting keypoints on vehicles.

Keypoint Detection timings

FAST	ORB	SHITOMASI	HARRIS	BRISK
1.47 ms	9 ms	15.8 ms	17.5 ms	40 ms

Descriptor extraction timings*

BRISK	BRIEF	ORB	SIFT	AKAZE
1.65 ms	2.5 ms	6.2 ms	49 ms	77 ms

Number of matches

Place	Combination
1st (186)	BRISK + BRIEF
2nd (180)	BRISK + SIFT
3rd (170)	BRISK + BRISK
4th (161)	BRISK + FREAK
5th (160)	BRISK + ORB
6th (139)	AKAZE + SIFT, AKAZE + AKAZE

7th (134)	AKAZE + BRISK
8th (132)	AKAZE + BRIEF
9th (130)	AKAZE + FREAK, AKAZE + ORB
10th (123)	FAST + AKAZE
11th (117)	FAST + SIFT, FAST + BRIEF
12th (115)	FAST + ORB

the top three Detector/Descriptor combinations are:

Place	Combination
1st place	BRISK + BRIEF (if prefer higher accuracy)
2nd place	FAST + BRIEF (if prefer speed)
3nd place	BRISK + BRISK (accuracy and speed are average level)