Code Explanation

MP1: code between line 73 : 77 in MidTermProject_Camera_Student.cpp

This code is inside the main for loop, when variable of the main for loop “imgIndex” will be 0 and 1 the dataBuffer will be already pushed twice “frame” into memory, so when “imgIndex” will be 2 this mean it will push a third place into memory means will contain frame 0, frame 1 and frame 2 and since the main tracking features is for the last two frames so putting condition of dataBuffer.size() will work when imageindex be 2 and make the vector of dataBuffer remove first element and push dataBuffer[1] to dataBuffer[0], after the condition is being done there push to the vector at position end()-1 which is push_back for the last frame, so the concept here is pop and push.

```
if (dataBuffer.size() >= dataBufferSize)
    {
     dataBuffer.erase(dataBuffer.begin());
    }
    dataBuffer.push_back(frame);
```

MP2: add the following keypoint detectors, code between line 338: 389 in matching2D.cpp

Here the implementation of selection the detector key point type as it’s required, firstly the parameters of this function the vector which will contain the keypoints of every frame from 0 to 9, the passing vector by reference in order to save the new key point to the original key points in the memory address not Permanent changes, img by reference for more speed load the matric image, string not big issue with using by value as well as bvis which is the bool for controlling the visualization key point at the end of this funcation.

Firstly, construct a pointer from datatype “cv::FeatureDetector” the name of pointer is “detector”, this pointer will contain the address of returned detector type from funcation cv::detectortype::create(), this will allow to have a way of accessing to a method of this class detector through pointer “detector”. If conditions will start to detect which detector type has been set up and passed to the function from user or main code through compare function and once any condition is work, it will start the previous procedure, all the detector type has been set the parameters as the default one except fast we set 30 instead of 10, the detector type here include (ORB,HARRIS, FAST, BRISK, SIFT, AKAZE), once the detector type selected, we passed to the detect() method and passed two arguments which is the frame image and where will put the extract all key point, so we put keypoint variable and since one of it's important evulation is to measure the time of extraction every keypoints in the frames, we measured the time of extraction and that why we start measuring time before detect() method exactly and stop measuring after it. The bvis bool is just to visualize the keypoints of every frame, but we make it off (false). Last thing we set this funcation to return type double this return the time of detect() for every frame, that will be used for Performance evulation on MP9.

```
double detKeypoints_Method(vector<cv::KeyPoint> &keypoints, cv::Mat &img, string detectorType, bool bVis)

{

    //I let the default parameters for every method


    cv::Ptr<cv::FeatureDetector> detector;


    if(detectorType.compare("SIFT") == 0) detector = cv::xfeatures2d::SIFT::create();


    if(detectorType.compare("BRISK") == 0) detector = cv::BRISK::create();


    if(detectorType.compare("FAST") == 0)

    {


    int threshold=30;


    detector = cv::FastFeatureDetector::create(threshold);


    }


   if(detectorType.compare("ORB") == 0) detector = cv::ORB::create();
```

```
    if(detectorType.compare("AKAZE") == 0) detector = cv::AKAZE::create();

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

    detector->detect( img, keypoints);

    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

    cout << detectorType << " Detector with n=" << keypoints.size() << " keypoints in " << 1000 * t / 1.0 <<
" ms" << endl;



    // visualize results
    if (bVis)
    {
        cv::Mat visImage = img.clone();

        cv::drawKeypoints(img, keypoints, visImage, cv::Scalar::all(-1),
cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

        string windowName = " Corner Detector Results of" +    detectorType;

        cv::namedWindow(windowName );

        imshow(windowName, visImage);

        cv::waitKey(0);
    } return t;}
```

The SHITOMASI is a sperate function due to it’s own way that cannot be combine with other method, I would do it at the previous function using some conditions to turn on, but I saw, It will be more good to leave as it’s located. The functions take same parameters as the previous function. I set the variable parameters as it was mentioned from previous lessons, the actual processing for detection key will be started, so I turn on the counter clock, I passed to cv::goodFeaturesToTrack() function which take the frame + vector which will save the keypoint in form point2f where point2f save every keypoint in form x and y coordinate { {x_0,y_0},{x_1,y_1} } , once the funcation done its own functionality, the “corners” vector contain all key points, but since we need to include all key points in vector “keypoints” , so I did for loops that will be access to every point with coordinate x,y and store it, I used the followed way of accessing to x and y of every index in the “corners” and then converting to Point2f, but it can be also following other way of access directly through the main index *(it) ={x_0,y_0}, the next important thing to add to key points attribute size then I pushed these data of every point to “keypoint” vector

Code is located in matching2D.cpp, line 147:213

```
double detKeypoints_SHITOMASI(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)

{



    // compute detector parameters based on image size

    int blockSize = 4;       //  size of an average block for computing a derivative covariation matrix over each pixel neighborhood


    double maxOverlap = 0.0; // max. permissible overlap between two features in %


    double minDistance = (1.0 - maxOverlap) * blockSize;


    int maxCorners = img.rows * img.cols / max(1.0, minDistance); // max. num. of keypoints


    double qualityLevel = 0.01; // minimal accepted quality of image corners

    double k = 0.04;


    // Apply corner detection
    //double t = (double)cv::getTickCount();

    vector<cv::Point2f> corners;

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

    cv::goodFeaturesToTrack(img, corners, maxCorners, qualityLevel, minDistance, cv::Mat(), blockSize,
false, k);

    // add corners to result vector
    for (auto it = corners.begin(); it != corners.end(); ++it)
    {

        cv::KeyPoint newKeyPoint;

        newKeyPoint.pt = cv::Point2f((*it).x, (*it).y); // you can use also *(it)

        newKeyPoint.size = blockSize;

        keypoints.push_back(newKeyPoint);

    }

    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

    cout << "Shi-Tomasi detection with n=" << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << "
ms" << endl;
```

```
if(bVis)
{
    cv::Mat visImage = img.clone();

    cv::drawKeypoints(img, keypoints, visImage, cv::Scalar::all(-1),
                      cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

    string windowName = "Shi-Tomasi Corner Detector Results";

    cv::namedWindow(windowName);

    imshow(windowName, visImage);

    cout << "Press any key to continue\n";

    cv::waitKey(0);
  }

  return t;

}
```

Code is inside matching2D.cpp line code form 218 : 334

After setting the parameters of cornerharris () which is block, apertureSize,minResponse and K =0.4 : 0.6, we turn on the timer counter from this point and then we extract the corner and inserted in image "dst' then we did normalize to normlize the range of pixel intensity ,so we set the range as in image gray 0 to 255 with adding the type of output image to CV_32F and then make scale with making sure that every element will be positive and not exceed 255 since it's 8bits range. Until here the "dst_norm_scaled" will display all key points, but all key points even if it's intersected with others, therefore we did two stages in order to get the uninterested corners or key points as well as a target intensity key points "threshold". So, we did nest for loop to check every intensity for every pixel and if the intensity is equal or higher 100 we saved into key point vector with define size of block and the save it. the second stage is to select from the selected first stage the key points that is not intersected then push it into keypoint and if even intersected then selected which one is more intensity and get out and then start the nested loop and repat the process.

```
double Harris_keypoints(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)

{

  int blockSize = 2;


  int apertureSize = 3;


  int minResponse = 100;


  double k = 0.04;




  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);

if(bVis)
{

string windowName = "Harris Corner Detector Response Matrix";

cv::namedWindow(windowName);

cv::imshow(windowName, dst_norm_scaled);

cv::waitKey(0);

}



double maxOverlap = 0.0;

for (size_t j = 0; j < dst_norm.rows; j++)
{

   for (size_t i = 0; i < dst_norm.cols; i++)
   {
```

```
int response = (int)dst_norm.at<float>(j, i);

if (response > minResponse)
{

    cv::KeyPoint newKeyPoint;

    newKeyPoint.pt = cv::Point2f(i, j);

    newKeyPoint.size = 2 * apertureSize;

    newKeyPoint.response = response;

    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();

    cout << "Haaris detection with n=" << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << " ms"
<< endl;




if(bVis)
{

string windowName = "Harris Corner Detection Results";
```

```
    cv::namedWindow(windowName);

    cv::Mat visImage = dst_norm_scaled.clone();

    cv::drawKeypoints(dst_norm_scaled, keypoints, visImage,  cv::Scalar::all(-1),
cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

    cv::imshow(windowName, visImage);

    cv::waitKey(0);

    }

   return t;

}
```

MP3: only keep keypoints on the preceding vehicle. Code is in MidTermProject_Camera_Student.cpp line 146 to 156

Here we started to create variable from datatype KeyPoint called “Keys_In_Rect” where all the key points within the specific rectangle is already located in constructor Rect with object vehicleRect. I set for loop in range where the key will contain key point elements and through attributes “pt” every key point coordinate will be entered as arguments and then if it’s inside the rectangle, it will be pushed these keypoints within Rectangle into vector “Keys_In_Rect”. After all points are collected, we replace the old points in “keypoints” vector to the points that inside “Kets_In_Rect” vector through assign() function, therefore before these line keypoints vector has all keypoints, but in line code 156 keypoints has only the keypoints within the Rectangle

----

```
vector<cv::KeyPoint> Keys_In_Rect ;

    bool bFocusOnVehicle = true;

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

    if (bFocusOnVehicle)

    {

      for(auto key: keypoints){


          if( vehicleRect.contains(key.pt)) Keys_In_Rect.push_back(key);


       }


     Keypoints.assign(Keys_In_Rect.begin(),Keys_In_Rect.end());


    }
```

MP4: Code is inside matching2D.cpp, line code from 82:141

In this function we extracted the descriptor with respect to the key points, in order to extract descriptor we have to construct pointer from data type cv::DescriptorExtractor this will point to the address of class where we can get access to method which is compute(). After we have been set the descriptor which at this case BRISK, BRIEF, ORB, FREAK, SIFT. The create will return address which will be stored in descriptor pointer and then as explained getting access to compute that will give us the descriptor data. We set the counter before and after exactly the compute function because this is the real time of extracting the descriptor. This function will return double data type which the time of this process for every frame.

```
double 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);

   }

   if (descriptorType.compare("BRIEF") == 0)

   {


     extractor = cv::xfeatures2d::BriefDescriptorExtractor::create();

   }
```

```
 if (descriptorType.compare("ORB") == 0)
{

   extractor = cv::ORB::create();
}



  if (descriptorType.compare("FREAK") == 0)
{

    extractor = cv::xfeatures2d::FREAK::create();
}



if (descriptorType.compare("AKAZE") == 0)
{

    extractor = cv::AKAZE::create();
}



if (descriptorType.compare("SIFT") == 0)
{

    extractor = cv::xfeatures2d::SIFT::create();
}
```

```
    // perform feature description
    double t = (double)cv::getTickCount();

    extractor->compute(img, keypoints, descriptors);

    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

    cout << descriptorType <<  1000 * t / 1.0 << " ms " << endl;

    return t ;
}
```

MP5: Code is exist in matching2D.cpp. code from 23 : 35

Afet matching which type of keypoints and descriptor matching will be used if “MAT_FLANN. The first thing is to check about the image type format, so check the type CV_32F is a critical with way of matches, so we set this condition that will be achieved incase type image is not CV_32F, then we used methodconverTo() which will convert the type of image to CV_32F. we set a pointer “matcher “ to point to the returned address of cv::DescriptorMatcher::create()

```
  else if (matcherType.compare("MAT_FLANN") == 0)

  {

    if (descSource.type() != CV_32F)

    {

      descSource.convertTo(descSource, CV_32F);


      descRef.convertTo(descRef, CV_32F);

    }


    matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);


    cout << "FLANN matching";

  }


  // perform matching task
```

MP6: Code is inside matching2D, code line from 38 : 66

Knn is another way matching key points. Firstly we construct vector vector of vector from data type of cv::DMatch { { keypoint_0_frame_1, keypoint_1_frame_1} } from source image. Time has been started to count the process of finding best two descriptor or image source and once knnmatch() terminated the counter stopped, where knnMatch it does find the best two descriptor on image resource and since we need only one we set for loop to make difference for the best fit with distance ratio and we pushed to matches variable.

```
 else if (selectorType.compare("SEL_KNN") == 0)

 {


   vector<vector<cv::DMatch>> knn_matches;


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


    matcher->knnMatch(descSource, descRef, knn_matches, 2); // finds the 2 best matches

   t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();


    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]);

        }

    }


 }
```

```
    cout << "Matching Type :" << matcherType <<" Selector type :" << selectorType <<" Detector with n="
<<matches.size() << " Match_Keypoints in " << 1000 * t / 1.0 << " ms" << endl;


  return t;

}
```