Evalu8

Autonomous Image Analysis of Drone Footage to Evaluate Visual Perception Load & Clutter Using C++ & OpenCV

Vidur Prasad
vidurtprasad@gmail.com
Dayton Regional STEM School

Kevin Durkee & John Feeney

APTIMA®
Human-Centered Engineering

Evalu8 Technical Manual

INTRODUCTION

Evalu8 addresses the growing need to bring in objective context to the analysis of cognitive workload while performing surveillance tasks through Unmanned Aerial Vehicles. Evalu8 analyzes drone footage and applies multiple feature extraction and optical flow algorithms to develop a rating of the visual clutter or load caused by the footage, using open source algorithms. Open Computer Vision (OpenCV) algorithms, version 2.4.11, developed by Intel, on a C++ platform, were utilized to create the application. This manual outlines the different algorithms that were used, and then briefly discus their implementation. The manual also outlines the future of the work, in terms of optimization systems, as well as investigating further correlations with already existing work in the field of visual perception load.

PART 1: READING IN FRAMES FOR ANALYSIS

The first task of Evalu8 is to read in frames, from common video formats, such as MP4, and save the frame in a format that allows for easy processing. Evalu8 does this by using the `VideoCapture API` that allows a file stream to be instantiated, and then various methods tap into this live stream. The individual frames are capture using the `.read` method to receive the current frame in a Matrix object format. The Matrix object (`Mat`), a type implemented by OpenCV, is then saved into a vector of Matrix objects, allowing for access to past frames. This system is also useful, as all of the multiple algorithms can each tap into the vector and read relevant objects, without the need to increase number of instances a frame needs to be stored, thereby reducing RAM usage. The frames in Evalu8 are stored in `vector <Mat> frames & vector <Mat> grayFrames`, containing color and gray scale frames respectively. The objects in these vectors can be accessed using `frames.at("index of Mat")` command. Since C(++) does not have memory management, one must be very careful in saving all of the frames as RAM usage can radically increase. To deal with this issue, Evalu8 cleans and deletes all frames more than .2 seconds old to reduce RAM usage. The frames are read in through the main while loop, and continue to be read in until all of the frames are read.

PART 2: PERFORMING FEATURE EXTRACTION

Four different algorithms are used to examine the complexity of the image through the analysis of features. The number of specific features, such as corners, and contours, are calculated using these four methods, and can therefore be used to build a model of the complexity of the image.

The first algorithm that is used is the Speed Up Robust Feature (SURF) algorithm, which is an algorithm that is based on the Hessian matrix detector, and is designed to be as light and efficient as possible. In OpenCV, SURF

is already implemented, and only takes in the Hessian value. This value range in the 400-800, and was set at 500 after analyzing its' affect on performance. SURF writes all key points into a vector of key points, and then draws these points onto an image. The SURF metric that is saved is the number of key points identified in the image.

The second algorithm that is used is the Shi-Tomasi corners detector, based on the Harris matrix, utilizing multiple methods to identify corners throughout the image. Evalu8 does not use the Harris detector through Shi-Tomasi, but uses Harris as a separate metric. The minimum quality level, or differential in quality between the best and the worst, is set rather low, at .1, to allow for a greater flexibility in scores, and therefore show differential between "easy" and "hard" images in a more explicit way. The minimum distance between corners is set at 10, and the block size, or size of the matrix being investigated is 3 X 3, and the constant k is set to .04. K is a free parameter that is used in the Harris matrix. The Shi-Tomasi method, otherwise known as the Good Features to Track Method, reads in gray scale frames to perform analysis. By identifying the number of corners detected, another metric on the visual clutter is collected.

The third algorithm that is used is the Harris corner detection system. The Harris algorithm utilizes the Eigen detector, receiving parameters about the aperture size to use, 3, the block size to analyze, 3, and to use the default border detection method. After the Eigen corner detector is run, the Harris detector is used to threshold the corners to only identify those that hit the quality levels required by the Harris algorithm. The number of Harris corners identified, found by processing through every pixel individually, is then returned.

The fourth algorithm that is used is the Canny contour detector, to identify the number of defined objects in the image. The Gaussian is first applied to blur the image, and then various thresholds are passed to try to identify defined contours. This algorithm has some issues, specifically, when there are trees, or other similar objects that contain lots of noise. The Canny detector identifies all of the sub-components, such as leaves, as discrete objects, dramatically inflating its values. To counter this, the Canny detector is weighed less in the final rating system. In addition, the Canny detector is mainly used to identify the noise that occurs in the drone footage.

PART 3: PERFORMING OPTICAL FLOW ANALYSIS

Farneback Dense Optical Flow Analysis (FDOFA) is performed to quantify all movement on screen, caused by the drone movement, and to provide another measure of context to the drone pilots' performance. FDOFA is based on the principle that clusters of pixels can be tracked over multiple frames to identify movement, and to there by determine the overall movement throughout the frame. This can be used to determine the average movement for each pixel, a metric that has a drastic impact on the accuracy of a pilot, as movement makes it much harder to track objects. FDOFA is implemented to use gray scale images, which have one channel, to

track movement. FDOFA uses multiple arguments to adjust the algorithm; the pyramid scale that is used is .5, causing a normal pyramid to be built; three pyramid levels are created; the windows size is 15, which allows for a good compromise between dealing with image noise, and performing a sharp, accurate OFA. In addition, three iterations are performed to optimize performance and five pixels are looked at nearby to find the polynomial expansion of each pixel. The standard deviation from the Gaussian is 1.2. After a vector of vectors for every pixels movement is generated, the sum of this is created. The first element of the sum is the total movement, and is the quantifiable value used to express the amount of motion. However, this value varies widely, from the low hundreds to almost a million. The absolute value is taken to ensure that only the magnitude, not the direction of movement, is analyzed. The normalization process is able to handle these values, however, the first time a spike occurs, even with normalization, the value spikes. To handle this, a threshold of 10000 is set, where, if crossed, a constant value of 100 is assigned to FDOFA. Due to the range of values, FDOFA, in essence, acts as a Boolean, showing when scene changes occur, providing greater context to Evalu8.

Printing multiple vectors over the image and displaying the movement through the vectors creates a graphical representation of FDOFA. This process is one of the most computationally taxing, and it is predicted that this is the greatest cause of performance lag in the system.

PART 4: COMPUTING THE FINAL SCORE

The final step in the Evalu8 process is calculating the final visual perception score. The final score is a metric from 0-100, that is an objective metric that quantifies the visual load on the human, and will give context to other metrics such as the TLX measure collected. The final visual perception score is calculated by keeping a running vector for every metric that stores every frame's values. Evalu8 processes ratings and attempts to build an accurate model by normalizing in real time, as well as in the end of processing. The final, post-processing, is a far more accurate method of normalization, and are the values that should be used when performing correlation. When the method is called, Evalu8 begins by normalizing all metric values, as well as all final score values. This is done by first determining the minimum and maximum values of the series. Values are then normalized by subtracting the minimum number from the current number, and then dividing by the range of the series. This normalizes the values between 0 and 1, and multiplying the values by 100 returns values ranging from 0 to 100. Values are also put through multiple safety checks, as some algorithms, such as Harris, have an affinity for returning bogus values in specific situations, and the algorithm is smart enough to threshold out these values, and can set them to a constant value. Care must be taken in future versions to account for errors specific to individual algorithms. SURF, Harris, and Shi-Tomasi are weighed at 3 times the level of FDOFA and Canny, as they are less temperamental, and offer a much better picture of the underlying image. These different methods, in conjunction, create a method that gets more accurate the longer it is used. Post-processing

also occurs by moving through all frames and deriving all of the ratings by using fully normalized values for each of the different metrics. These values are then printed to a txt file, with the frame number, space, and final rating on one line.

PART 5: OUTPUTTING DATA

The system is designed to run read frames and display them, and the various algorithms running to allow the operator to gain an understanding of the analysis that each algorithm is performing. The different feeds are output to different windows, if the `displayWindows()` method is used, and all of the data is written to an image and then output for the operator to see. The normalized values for all five algorithms, the frame number, and the final rating are all outputted using the `putText()` method. All of the data, except the frames, which are cleaned after a buffer of 10 frames are created, is stored in vectors. At the end of execution, or if the program is forced to stop, with a double tap of the escape key, all of the data, all of the feature data, is written to a txt file for each frame, allowing for future analysis. Outputting the data to image slows down the process but not significantly. A file containing all data, frame number, number of SURF features, number of Harris corners, number of Shi-Tomasi corners, number of Canny contours, sum of Farneback dense optical flow analysis, and the frames per second at the time in plain text. This file has the name of the video file, time and date of runtime, and rawData.txt in the filename of the output file. Another file with one column of frame numbers, and one column of normalized final ratings, with one space in the middle, is also printed, with the same format as former file, but with finalRatingsNormalized.txt append. A file appended with Stats.txt is printed in the start with basic information about video.

MULTITHREADING

The current software is multithreaded, where each of the discrete image analysis algorithms run on individual threads to maximum resource allocation. Multithreaded software, however, cannot force systems to utilize multiple cores for each of the threads, however, on a MacBook Pro running OSX Yosemite, system utilization of two cores has been seen. The threads are opened in succession, run in tandem, and execution continues after all threads report that they are finished. To facilitate this system, global variables are necessary, as all threads can mutate these variables. The pthread library is used to multithread, a system designed for use on Unix systems. Emulators exist for Windows, but more careful analysis will need to occur.

KNOWN CHALLENGES / TROUBLESHOOTING

There are multiple issues, specifically with regard to memory leaks that occur with the system, as video requires high memory usage. Memory leaks, specifically mistakes causing objects to be erased, have occurred, and will cause the FDOFA to break, as the previous frames will not be accessible. This caused RAM usage to spike to

over 38 Gb during 10,000 frame tests. To eliminate this issue, frames older than approximately .12 seconds, or about 3 frames, are replaced with a pointer to a small matrix, radically reducing memory usage. This cleaning occurs at the end of a cycle, causing the RAM usage to fluctuate around 20 MB. The vectors are vectors of pointers, causing memory leaks to occur as objects can be left in the memory, although without a reference. To fix this, the current `frameToBeDisplayed` is specifically deleted from RAM after one cycle, and is then reloaded. All other objects are merely pointers. The nominal RAM usage is around 100 MB. A system to delete and clean all of the vectors on exit, as C++ does not have memory management, was also implemented to ensure that Evalu8 runs cleanly, and does not interfere with other programs.

Another known issue is the volatility of the Canny Edge Detector, specifically when viewing wildlife. However, as has been recognized that wildlife causes significant visual perception load, and since Canny is a good tool to detect noise, its weight has not been reduced.

A central counter is in place to help with global sequencing, but if the counter gets of, or if calls to retrieve data from vectors are not made exactly, then there is a high probability for vector out of range errors.

FDOFA frequently returns values that range to almost 10^313, causing overflows. To counteract this, various fail safe measures, such as carefully utilizing only the normalized version of these numbers, and ensuring that long versions of the variable is used were incorporated.

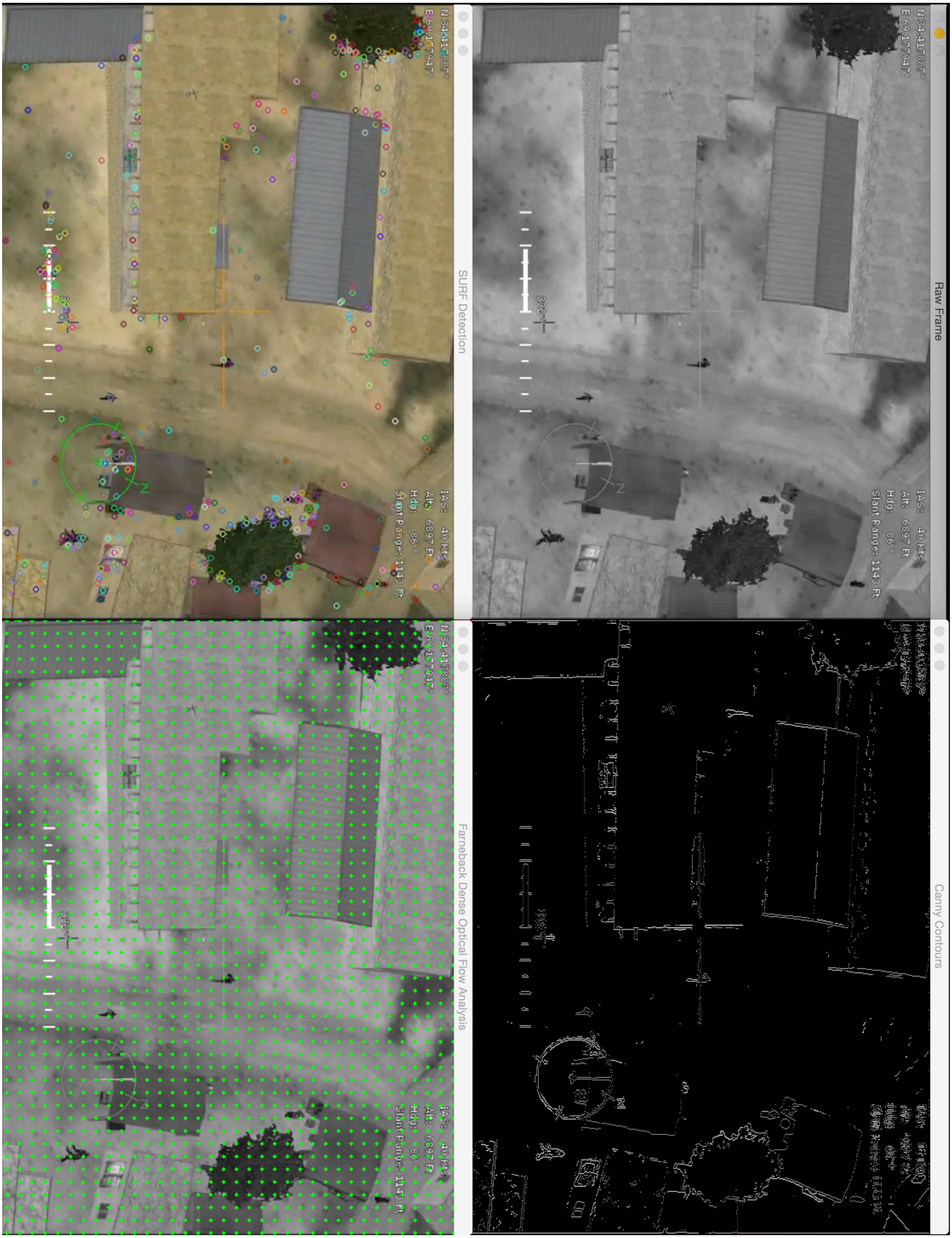
System processing degrades a little bit over time due to the increase in RAM usage and overhead, and timers and methods are in place to track statistics related to time, and for necessary processing enhancements to be tested.

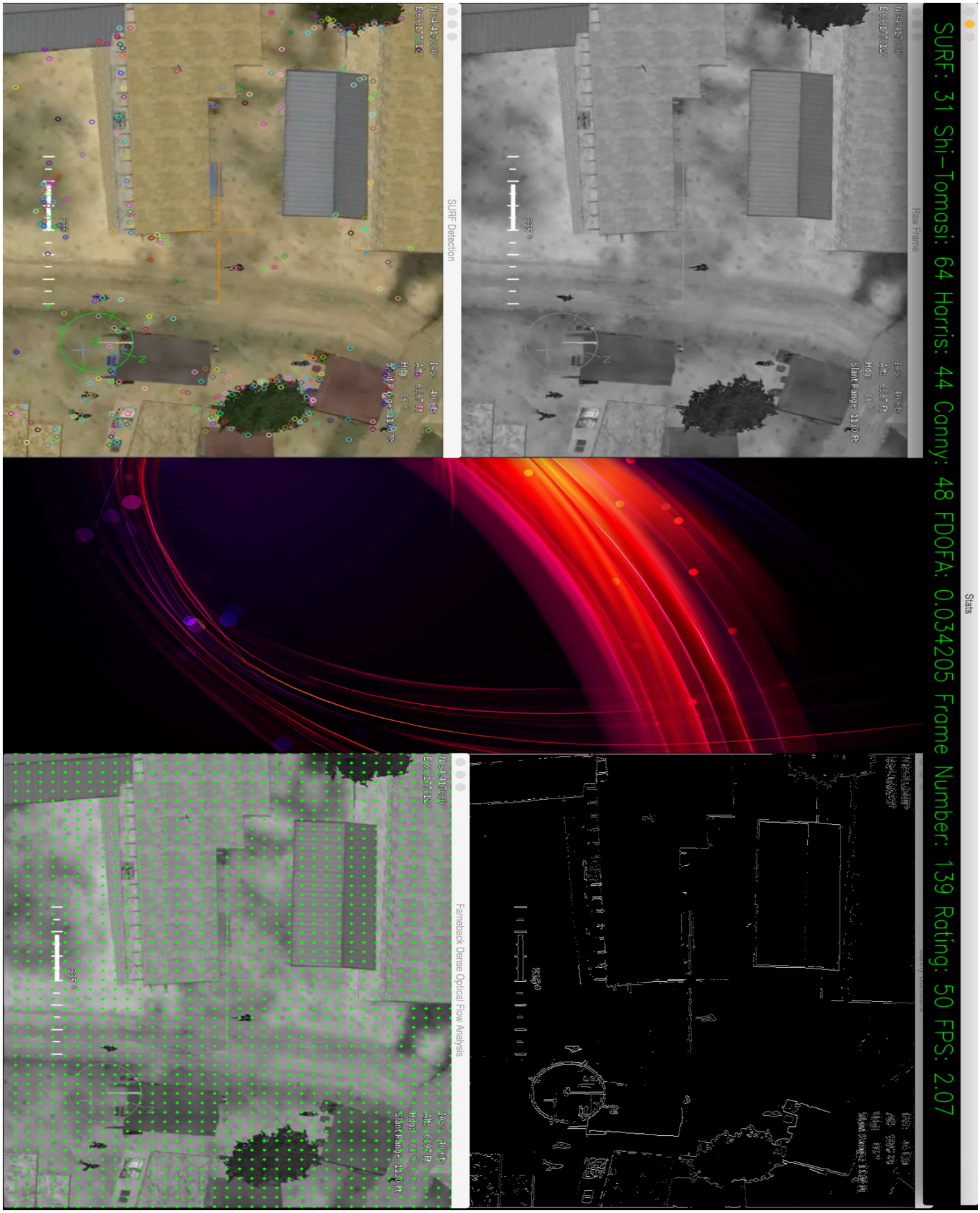
The frame rate, run on a MacBook Pro with Retina Display, with 16 Gb DDR3 RAM, 2.5 GHz Intel Core i7-2860QM, ranges from around 1.8 – 2.1 Frames per Second (FPS). As most videos at 25 frames per second, this will need to be increased. However, merely sampling a segment of frames, and not testing every frame can achieve similar results.

FUTURE WORK

Future work will focus on finding a correlation, or relationship between the calculated visual perception load and the recorded workload values calculated using the TLX method, as well as using physiological and performance based metrics collected in tandem with the simulated UAV footage. Work will need to occur, possibly using Machine Learning algorithms, to create a system that is able to correlate these factors, and to even use visual perception load as a factor in calculating load, or at least providing context. Future work will also need to occur to integrate Evalu8 with the existing infrastructure setup for the drone footage tests. As the existing MP4 videos are converted to video streams, it is possible to use a live stream of footage with Evalu8. In

addition, more work will also need to occur to bring down processing time. In conclusion, Evalu8 has a bright future in terms of aligning with Aptima, and by extension, the Air Force's goal to augment sensor data to get high-quality context by determining the visual perception load on drone pilots.


Raw Frame
SURF Detection
Canny Contours
Farneback Dense Optical Flow Analysis


SURF: 31 Shi-Tomasi: 64 Harris: 44 Canny: 48 FDOFA: 0.034205 Frame Number: 139 Rating: 50 FPS: 2.07
Stats
Raw Frame
SURF Detection
Farneback Dense Optical Flow Analysis

WORKS CITED

Bay, H. *SURF: Speeded Up Robust Features.* ETH Zurich, Zurich.

Farneback, G. *Two-Frame Motion Estimation Based on Polynomial Expansion.* Linkoping University, Computer Vision Labratory. Linkoping: Linkoping University.

Horn, B. K., & Schunck, B. G. *Determining Optical Flow.* Massachusetts Institute of Technology, Artifical Intelligence Laboratory. Boston: MIT.

Kim, S.-H. (2011). Multidimensional Measure of Display Clutter and Pilot Performance for Advanced Head-up Display. *Aviation, Space, and Enviromental Medicine , 82* (10), 1-10.

Rosenholtz, R. *Feature Congestion: A Measure of Display Clutter.* MIT , Brain & Cognitive Sciences, Cambridge.

Shi, J., & Carlo Tomasi. (1994). Good Features to Track. *IEEE Conference on Computer Vision and Pattern Recognition.* Seattle: IEEE.

Shrivakshan, G. T. (2012). A Comparison of various Edge Detection Techniques used in Image Processing. *International Journal of Computer Science , 9* (5), 269-276.

EVALU8 C++ W/ OPENCV CODE

```
//============================================================================
// Name        : Evalu8.cpp
// Author      : Vidur Prasad
// Version     : 1.5.0
// Copyright   : APTIMA Inc.
// Description : Autonomous Image Analysis of Drone Footage to Evaluate Visual Perception Load & Clutter
//============================================================================

//============================================================================
// Metrics Polled
//      Basic Number of Features --> SURF Detection
//      Number of Corners 1 --> Harris Corner Detection
//      Number of Corners 2 --> Shi-Tomasi Feature Extraction
//      Number of Edges --> Canny Edge Detector
//      Optical Flow Analysis --> Farneback Dense Optical Flow Analysis
//============================================================================

//include opencv library files
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/video/tracking.hpp>
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/gpu/gpu.hpp"
#include <opencv2/nonfree/ocl.hpp>

//include c++ files
#include <iostream>
#include <fstream>
#include <ctime>
#include <time.h>
#include <thread>
#include <chrono>
#include <stdio.h>
#include <stdlib.h>
#include <limits>
#include <math.h>
#include <algorithm>
#include <vector>
#include <pthread.h>
#include <cstdlib>

//declaring templates for use in max element function
template <typename T, size_t N> const T* mybegin(const T (&a)[N]) { return a; }
template <typename T, size_t N> const T* myend  (const T (&a)[N]) { return a+N; }

//namespaces for convenience
using namespace cv;
using namespace std;

//multithreading global variables
vector <Mat> globalFrames;
int FRAME_HEIGHT;
int FRAME_WIDTH;
Mat globalGrayFrame;
//setting constant filename to read form
const char* filename = "assets/P2-T1-V1-TCheck-Final.mp4";
//const char* filename = "assets/The Original Grumpy Cat!.mp4";
//const char* filename = "assets/P8_T5_V1.mp4";

//SURF Global Variables
static Mat surfFrame;
```

```
int surfThreadCompletion = 0;
int numberOfSURFFeatures = 0;

//canny Global Variables
int numberOfContoursThread = 0;
int cannyThreadCompletion = 0;
Mat cannyFrame;

//Shi Tomasi Global Variables
int shiTomasiFeatures = 0;
int shiTomasiThreadCompletion = 0;

//harris global variables
int numberOfHarrisCornersCounter = 0;
int harrisCornersThreadCompletion = 0;

//optical flow global variables
int sumOpticalFlow = 0;
int opticalFlowThreadCompletion = 0;
Mat optFlow;

//defining format of data sent to threads
struct thread_data{
   //include iteration number
   int i;
};

//method to draw optical flow, only should be called during demos
static void drawOptFlowMap(const Mat& flow, Mat& cflowmap, int step,
                    double, const Scalar& color)
{
        //iterating through each pixel and drawing vector
    for(int y = 0; y < cflowmap.rows; y += step)
        for(int x = 0; x < cflowmap.cols; x += step)
        {
            const Point2f& fxy = flow.at<Point2f>(y, x);
            line(cflowmap, Point(x,y), Point(cvRound(x+fxy.x), cvRound(y+fxy.y)),
                 color);
            circle(cflowmap, Point(x,y), 2, color, -1);
        }
}

//method that returns date and time as a string to tag txt files
const string currentDateTime()
{
        //creating time object that reads current time
    time_t now = time(0);
    //creating time structure
    struct tm tstruct;
    //creating a character buffer of 80 characters
    char buf[80];
    //checking current local time
    tstruct = *localtime(&now);
    //writing time to string
    strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);
    //returning the string with the time
    return buf;
}

//method to perform optical flow analysis
void *computeOpticalFlowAnalysisThread(void *threadarg)
{
        //reading in data sent to thread into local variable
        struct thread_data *data;
        data = (struct thread_data *) threadarg;
```

```
        int i = data->i;

        //defining local variables for FDOFA
        Mat prevFrame, currFrame;
        Mat gray, prevGray, flow,cflow;

        //reading in current and previous frames
        prevFrame = globalFrames.at(i-1);
        currFrame = globalFrames.at(i);

        //converting to grayscale
        cvtColor(currFrame, gray,COLOR_BGR2GRAY);
        cvtColor(prevFrame, prevGray, COLOR_BGR2GRAY);

        //calculating optical flow
        calcOpticalFlowFarneback(prevGray, gray, flow, 0.5, 3, 15, 3, 5, 1.2, 0);
        //converting to display format
        cvtColor(prevGray, cflow, COLOR_GRAY2BGR);
        //drawing optical flow vectors
        drawOptFlowMap(flow, cflow, 15, 1.5, Scalar(0, 255, 0));
        //saving to global variable for display
        optFlow = cflow;
        //returning sum of all movement in frame
        sumOpticalFlow = (abs(sum(flow)[0]));
        //signal thread completion
        opticalFlowThreadCompletion = 1;

}

//calculate number of Harris corners
void *computeHarrisThread(void *threadarg)
{
        //reading in data sent to thread into local variable
        struct thread_data *data;
        data = (struct thread_data *) threadarg;
        int i = data->i;

        //defining local variables for Harris
        numberOfHarrisCornersCounter = 0;
        int blockSize = 3;
        const int apertureSize = 3;
        double harrisMinValue;
        double harrisMaxValue;
        double harrisQualityLevel = 35;
        double maxQualityLevel = 100;

    //create frame formatted for use in Harris
    Mat harrisDST = Mat::zeros(globalGrayFrame.size(), CV_32FC(6) );
    Mat mc = Mat::zeros(globalGrayFrame.size(), CV_32FC1 );
    Mat harrisCornerFrame = globalGrayFrame;

    //run Corner Eigen Vals and Vecs to find corners
    cornerEigenValsAndVecs( globalGrayFrame, harrisDST, blockSize, apertureSize, BORDER_DEFAULT );

    //use Eigen values to step through each pixel individaully and finish applying equation
    for( int j = 0; j < globalGrayFrame.rows; j++ )
    {
        for( int h = 0; h < globalGrayFrame.cols; h++ )
        {
                //apply algorithm
                        float lambda_1 = harrisDST.at<Vec6f>(j, h)[0];
                        float lambda_2 = harrisDST.at<Vec6f>(j, h)[1];
                        mc.at<float>(j,h) = lambda_1*lambda_2 - 0.04f*pow( ( lambda_1 + lambda_2 ), 2 );
        }
    }
```

```
    //find locations of minimums and maximums
    minMaxLoc( mc, &harrisMinValue, &harrisMaxValue, 0, 0, Mat() );

    //apply harris properly to every pixel
    for( int j = 0; j < globalGrayFrame.rows; j++ )
    {
        for( int h = 0; h < globalGrayFrame.cols; h++ )
            {
                        if( mc.at<float>(j,h) > harrisMinValue + ( harrisMaxValue - harrisMinValue )*
harrisQualityLevel/maxQualityLevel)
                        {
                                //apply algorithm, and increment counters
                                numberOfHarrisCornersCounter++;
                        }
                }
        }

        //signal completion
        harrisCornersThreadCompletion = 1;

}

//calculate number of Shi-Tomasi corners
void *computeShiTomasiThread(void *threadarg)
{
        //reading in data sent to thread into local variable
        struct thread_data *data;
    data = (struct thread_data *) threadarg;
    int i = data->i;

        //defining local variables for Shi-Tomasi
        vector<Point2f> cornersf;
        const double qualityLevel = 0.1;
        const double minDistance = 10;
        const int blockSize = 3;
        const double k = 0.04;

        //harris detector is used seperately
        const bool useHarrisDetector = false;

        //setting max number of corners to largest possible value
        const int maxCorners = numeric_limits<int>::max();

        //perform Shi-Tomasi algorithm
    goodFeaturesToTrack(globalGrayFrame, cornersf, maxCorners, qualityLevel, minDistance,
    Mat(), blockSize, useHarrisDetector,k);

    //return number of Shi Tomasi corners
    shiTomasiFeatures = cornersf.size();

    //signal completion
    shiTomasiThreadCompletion = 1;
}

//calculate number of SURF features
void *computeSURFThread(void *threadarg)
{
        //reading in data sent to thread into local variable
   struct thread_data *data;
   data = (struct thread_data *) threadarg;
   int i = data->i;

   //setting constant integer minimum Hessian for SURF Recommended between 400-800
   const int minHessian = 500;
```

```
   //defining vector to contain all features
   vector <KeyPoint> vectKeyPoints;
   //saving global frame into surfFrame
   globalFrames.at(i).copyTo(surfFrame);

   //running SURF detector
   SurfFeatureDetector detector(minHessian);
   detector.detect(surfFrame, vectKeyPoints );

   //drawing keypoints
   drawKeypoints(surfFrame, vectKeyPoints, surfFrame, Scalar::all(-1), DrawMatchesFlags::DEFAULT );
   numberOfSURFFeatures = vectKeyPoints.size();

   //signal completion
   surfThreadCompletion = 1;
}

//calculate number of contours
void *computeCannyThread(void *threadarg)
{
        vector<Vec4i> hierarchy;
        typedef vector<vector<Point> > TContours;
        TContours contours;
        struct thread_data *data;
        data = (struct thread_data *) threadarg;
        int i = data->i;
        //run canny edge detector
        Canny(globalFrames.at(i), cannyFrame, 115, 115);
        findContours(cannyFrame, contours, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_NONE);
        //return number of contours detected
        //imshow("globalFrames", contours);

    numberOfContoursThread = contours.size();

    cannyThreadCompletion = 1;
}

//calculate mean of vector of ints
double calculateMeanVector(Vector <int> scores)
{
  double total;
  //sum all elements of vector
  for(int i = 0; i < scores.size(); i++)
  { total += abs(scores[i]); }
  //divide by number of elements
  return total / scores.size();
}

//calculate mean of vector of ints
double calculateMeanVector(vector <int> scores)
{
  double total;
  //sum all elements of vector
  for(int i = 0; i < scores.size(); i++)
  { total += abs(scores.at(i)); }
  //divide by number of elements
  return total / scores.size();
}


//calculate mean of vector of oubles
double calculateMeanVector(Vector <double> scores)
{
  double total;
  //sum all elements of vector
```

```
  for(int i = 0; i < scores.size(); i++)
  { total += abs(scores[i]); }
  //divide by number of elements
  return total / scores.size();
}

//method to save all metrics to file after processing
void saveToTxtFile(int FRAME_RATE, Vector <int> vectNumberOfKeyPoints, Vector <int>
numberOfShiTomasiKeyPoints, Vector <int>
numberOfContours, Vector <int> numberOfHarrisCorners, Vector <double> opticalFlowAnalysisFarnebackNumbers,
vector <string> FPS, const char* filename)
{
        //instantiating file stream
        ofstream file;

        //creating filename ending
        string vectFilenameAppend = " rawData.txt";

        //concanating and creating file name string
        string strVectFilename = filename + currentDateTime() + vectFilenameAppend;

        //creating file
        file.open (strVectFilename);

        //save txt file
        for(int v = 0; v < vectNumberOfKeyPoints.size() - 5; v++)
        {
                file << "Frame Number " << v << " at " << (v * (1.0 / FRAME_RATE)) << " seconds has ";
                file << vectNumberOfKeyPoints[v];
                file << " SURF key points & ";
                file << numberOfShiTomasiKeyPoints[v];
                file << " Shi-Tomasi key points";
                file << " & " << numberOfContours[v] << " contours & ";
                file << numberOfHarrisCorners[v];
                file << " Harris Corners & FDOFA is ";
                file << opticalFlowAnalysisFarnebackNumbers[v];
                file << ". FPS is ";
                file << FPS.at(v);
                file << ".\n";
        }

        //close file stream
        file.close();
}

//save final ratings to text file
void saveToTxtFile(vector <int> finalRatings, string vectFilenameAppend)
{
        //instantiate new filestream
        ofstream file;

        //concanating and creating file name string
        string strVectFilename = filename + currentDateTime() + vectFilenameAppend;

        //create file
        file.open (strVectFilename);

        //save txt file
        for(int v = 0; v < finalRatings.size() ; v++)
        {
                file << v << " " << finalRatings.at(v) << endl;
        }

        //close file stream
        file.close();
```

```
}


//method to calculate tootal run time
void computeRunTime(clock_t t1, clock_t t2, int framesRead)
{
	//subtract from start time
	float diff ((float)t2-(float)t1);

	//calculate frames per second
	double frameRateProcessing = (framesRead / diff) * CLOCKS_PER_SEC;

	//display amount of time for run time
	cout << (diff / CLOCKS_PER_SEC) << " seconds of run time." << endl;

	//display number of frames processed per second
	cout << frameRateProcessing << " frames processed per second." << endl;
	cout << framesRead << " frames read." << endl;
}

//calculate time for each iteration
double calculateFPS(clock_t tStart, clock_t tFinal)
{
	//return frames per second
	return 1/((((float)tFinal-(float)tStart) / CLOCKS_PER_SEC));
}

//write initial statistics about the video
void writeInitialStats(int NUMBER_OF_FRAMES, int FRAME_RATE, int FRAME_WIDTH, int FRAME_HEIGHT, const
char* filename)
{
	////writing stats to txt file
	//initiating write stream
	ofstream writeToFile;

	//creating filename ending
	string filenameAppend = "Stats.txt";

	//concanating and creating file name string
	string strFilename = filename + currentDateTime() + filenameAppend;

	//open file stream and begin writing file
	writeToFile.open (strFilename);

	//write video statistics
	writeToFile << "Stats on video >> There are = " << NUMBER_OF_FRAMES << " frames. The frame rate is " <<
FRAME_RATE
	<< " frames per second. Resolution is " << FRAME_WIDTH << " X " << FRAME_HEIGHT;

	//close file stream
	writeToFile.close();

	//display video statistics
	cout << "Stats on video >> There are = " << NUMBER_OF_FRAMES << " frames. The frame rate is " <<
FRAME_RATE
	<< " frames per second. Resolution is " << FRAME_WIDTH << " X " << FRAME_HEIGHT << endl;;

}

//normalize vector values in real time
int realTimeNormalization(Vector <int> vectorToNormalize, int i)
{
	//determine max and min values
	double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
	double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());
```

```
	//perform normalization and return values
	return (((vectorToNormalize[i] - minElement) / (maxElement - minElement))*100);

}

//normalize vector values in real time
int realTimeNormalization(vector <int> vectorToNormalize, int i)
{
	//determine max and min values
	double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
	double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());

	//perform normalization and return values
	return (((vectorToNormalize.at(i) - minElement) / (maxElement - minElement))*100);

}

//normalize vector values in real time
double realTimeNormalization(Vector <double> vectorToNormalize, int i)
{
	//determine max and min values
	double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
	double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());

	//perform normalization and return values
	return (((vectorToNormalize[i] - minElement) / (maxElement - minElement))*100);

}

//normalize vector values in real time
double realTimeNormalization(vector <double> vectorToNormalize, int i)
{
	//determine max and min values
	double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
	double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());

	//perform normalization and return values
	return (((vectorToNormalize[i] - minElement) / (maxElement - minElement))*100);

}

//method to compute raw final score and recieve vectors of metrics
int computeFinalScore(Vector <int> vectNumberOfKeyPoints,Vector <int> numberOfHarrisCorners,
	Vector <int> numberOfShiTomasiKeyPoints, Vector <int> numberOfContours, Vector <double>
opticalFlowAnalysisFarnebackNumbers, int i)
{
	//normalize and weigh appropriately
	double numberOfKeyPointsNormalized = abs(3 * realTimeNormalization(vectNumberOfKeyPoints,i));
	double numberOfShiTomasiKeyPointsNormalized = abs(3 *
realTimeNormalization(numberOfShiTomasiKeyPoints,i));
	double numberOfHarrisCornersNormalized = abs(3 * realTimeNormalization(numberOfHarrisCorners,i));
	double numberOfContoursNormalized = abs(1 * realTimeNormalization(numberOfContours,i));
	double opticalFlowAnalysisFarnebackNumbersNormalized = abs((1 *
realTimeNormalization(opticalFlowAnalysisFarnebackNumbers,i)));

	//if FDOFA normalization fails
	if(opticalFlowAnalysisFarnebackNumbersNormalized > 1000)
	{
		//set FDOFA to tmp value
		opticalFlowAnalysisFarnebackNumbersNormalized = 100;
	}

	//determine final score by summing all values
	long int finalScore = abs(((numberOfKeyPointsNormalized + numberOfShiTomasiKeyPointsNormalized +
```

```
                        numberOfHarrisCornersNormalized + numberOfContoursNormalized +
opticalFlowAnalysisFarnebackNumbersNormalized))
                                );

        //return final score
        return finalScore;
}

//normalize vector in postprocessing
vector <int> normalizeVector(vector <int> vectorToNormalize)
{
        //declaring vector to store normalized score
        vector <int> normalizedValues;

        //int maxElement = max_element(begin(finalScores), end(finalScores));
        double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
        double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());

        //normalize every value
        for(int i = 0; i < vectorToNormalize.size(); i++)
        {
                normalizedValues.push_back(((vectorToNormalize.at(i) - minElement) / (maxElement -
minElement))*100);
        }

        //return normalized vector
        return normalizedValues;
}

vector <int> normalizeVector(Vector <int> vectorToNormalize)
{
        //declaring vector to store normalized score
        vector <int> normalizedValues;

        //int maxElement = max_element(begin(finalScores), end(finalScores));
        double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
        double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());

        //normalize every value
        for(int i = 0; i < vectorToNormalize.size(); i++)
        {
                normalizedValues.push_back(((vectorToNormalize[i] - minElement) / (maxElement -
minElement))*100);
        }

        //return normalized vector
        return normalizedValues;
}

//normalize vector in postprocessing
vector <double> normalizeVector(Vector <double> vectorToNormalize)
{
        //declaring vector to store normalized score
        vector <double> normalizedValues;

        //int maxElement = max_element(begin(finalScores), end(finalScores));
        double maxElement = *max_element(vectorToNormalize.begin(), vectorToNormalize.end());
        double minElement = *min_element(vectorToNormalize.begin(), vectorToNormalize.end());

        //normalize every value
        for(int i = 0; i < vectorToNormalize.size(); i++)
        {
                normalizedValues.push_back(((vectorToNormalize[i] - minElement) / (maxElement -
minElement))*100);
        }
```

```
	//return normalized vector
	return normalizedValues;
}

//normalize all ratings in post processing
void normalizeRatings(Vector <int> vectNumberOfKeyPoints, Vector <int> numberOfShiTomasiKeyPoints, Vector
<int> numberOfHarrisCorners,
	Vector <int> numberOfContours, Vector <double> opticalFlowAnalysisFarnebackNumbers)
{
	//declaring vector to store normalized score
	vector <int> finalScoreNormalized;

	//normalize all metric vector
	vector <int> vectNumberOfKeyPointsNormalized = normalizeVector(vectNumberOfKeyPoints);
	vector <int> numberOfShiTomasiKeyPointsNormalized = normalizeVector(numberOfShiTomasiKeyPoints);
	vector <int> numberOfHarrisCornersNormalized = normalizeVector(numberOfHarrisCorners);
	vector <int> numberOfContoursNormalized = normalizeVector(numberOfContours);
	vector <double> opticalFlowAnalysisFarnebackNumbersNormalized =
normalizeVector(opticalFlowAnalysisFarnebackNumbers);

	//calculate score for each frame
	for(int i = 11; i < vectNumberOfKeyPoints.size(); i++)
	{
		double score = vectNumberOfKeyPointsNormalized.at(i) * 3 +
	numberOfShiTomasiKeyPointsNormalized.at(i) * 3 +
		numberOfHarrisCornersNormalized.at(i) * 3 + numberOfContoursNormalized.at(i) +
opticalFlowAnalysisFarnebackNumbersNormalized.at(i-11);
		score /= 11;
		finalScoreNormalized.push_back(score);
	}

	//normalize final ratings
	finalScoreNormalized = normalizeVector(finalScoreNormalized);

	//save normalized final score
	saveToTxtFile(finalScoreNormalized, " finalRatingsNormalized.txt");

}

//display all windows
void displayWindows(int i)
{
	//if all frames have data
	if(i > 12)
	{
		imshow("Raw Frame", globalGrayFrame);
		imshow("SURF Detection", surfFrame);
		imshow("Canny Contours", cannyFrame);
		//check optical flow section
		imshow("Farneback Dense Optical Flow Analysis", optFlow);
	}
}


//method to close all windows
void destroyWindows()
{
	//close windows
	destroyWindow("Raw Frame");
	destroyWindow("SURF Detection");
	destroyWindow("Canny Contours");
	destroyWindow("Farneback Dense Optical Flow Analysis");
}
```

```
//main method
int main() {

        //display welcome image
        imshow("Welcome", imread("assets/Aptima.jpg"));

        //put thread to sleep until user is ready
        this_thread::sleep_for (std::chrono::seconds(5));

        //close welcome image
        destroyWindow("Welcome");

        //creating initial and final clock objects
        //taking current time when run starts
        clock_t t1=clock();

        //random number generator
        RNG rng(12345);

        //defining VideoCapture object and filename to capture from
        VideoCapture capture(filename);

        //declaring strings for all metrics
    string strRating, strNumberOfHarrisCorners, strNumberOfShiTomasiCorners, numberOfKeyPointsSURF, strCanny,
strActiveTimeDifference;

    //initializing string to display blank
        string strDisplay =  "";
        string strNumberOpticalFlowAnalysis = "";

        //creating vectors to store all metrics
        vector <int> numberOfContours;
        vector <int> numberOfShiTomasiKeyPoints;
        vector <int> numberOfHarrisCorners;
        vector <double> opticalFlowAnalysisFarnebackNumbers;
        vector <int> vectNumberOfKeyPoints;
        vector <int> finalScores;
        vector <String> FPS;

        //collecting statistics about the video
        //constants that will not change
        const int NUMBER_OF_FRAMES =(int) capture.get(CV_CAP_PROP_FRAME_COUNT);
        const int FRAME_RATE = (int) capture.get(CV_CAP_PROP_FPS);
        FRAME_WIDTH = capture.get(CV_CAP_PROP_FRAME_WIDTH);
        FRAME_HEIGHT = capture.get(CV_CAP_PROP_FRAME_HEIGHT);

        writeInitialStats(NUMBER_OF_FRAMES, FRAME_RATE, FRAME_WIDTH, FRAME_HEIGHT, filename);

        // declaring and initially setting variables that will be actively updated during runtime
        int framesRead = (int) capture.get(CV_CAP_PROP_POS_FRAMES);
        double framesTimeLeft = (capture.get(CV_CAP_PROP_POS_MSEC)) / 1000;

        //initializing counters
        int i = 0;

        //creating placeholder object
        Mat placeHolder = Mat::eye(1, 1, CV_64F);

        //actual run time, while video is not finished
        while(framesRead < NUMBER_OF_FRAMES)
        {
                clock_t tStart = clock();

                //create pointer to new object
                Mat * frameToBeDisplayed = new Mat();
```

```
            //reading in current frame
            capture.read(*frameToBeDisplayed);

            //adding current frame to vector/array list of matricies
            globalFrames.push_back(*frameToBeDisplayed);

            //convert frame to grayscale
            cvtColor(globalFrames.at(i), globalGrayFrame, CV_BGR2GRAY);

            //instantiating multithread objects
            pthread_t surfThread;
            pthread_t cannyThread;
            pthread_t shiTomasiThread;
            pthread_t harrisThread;
            pthread_t opticalFlowThread;

            //instantiating multithread Data object
            struct thread_data threadData;

            //saving data into data object
            threadData.i = i;

        //creating threads
        int surfThreadRC = pthread_create(&surfThread, NULL, computeSURFThread, (void *)&threadData);
        int cannyThreadRC = pthread_create(&cannyThread, NULL, computeCannyThread, (void *)&threadData);
            int shiTomasiRC = pthread_create(&shiTomasiThread, NULL, computeShiTomasiThread, (void
*)&threadData);
            int harrisRC = pthread_create(&harrisThread, NULL, computeHarrisThread, (void *)&threadData);

        //check if all threads created
        if (surfThreadRC || cannyThreadRC || shiTomasiRC || harrisRC)
        {
            //throw error
            throw "Error:unable to create thread";

            //exit if issue
            exit(-1);
        }

            //if ready for FDOFA
            if(i > 10)
            {
                    int opticalFlowRC = pthread_create(&opticalFlowThread, NULL,
computeOpticalFlowAnalysisThread, (void *)&threadData);

                    if (opticalFlowRC)
                    {
                            cout << "Error:unable to create thread," << opticalFlowRC << endl;
                            exit(-1);
                    }
            }

            //check if OFA is being performed
            if(i<= 10)
            {
                    //idle until all threads finished
                    while(surfThreadCompletion == 0 || cannyThreadCompletion == 0 ||
shiTomasiThreadCompletion == 0 || harrisCornersThreadCompletion == 0)
                    {
                    }
            }
            else
            {
                    //idle until all threads finished
```

```
                        while(surfThreadCompletion == 0 || cannyThreadCompletion == 0 ||
                                                                shiTomasiThreadCompletion == 0 ||
harrisCornersThreadCompletion == 0 || opticalFlowThreadCompletion == 0)
                        {
                        }
                }

                //writing that all threads are ready for next run
                shiTomasiThreadCompletion = 0;
                surfThreadCompletion = 0;
                cannyThreadCompletion = 0;
                harrisCornersThreadCompletion = 0;
                opticalFlowThreadCompletion = 0;

                //write Canny
                numberOfContours.push_back(numberOfContoursThread);
                String strCanny = to_string(realTimeNormalization(numberOfContours, i));

                //write ShiTomasi
                numberOfShiTomasiKeyPoints.push_back(shiTomasiFeatures);
                String strNumberOfShiTomasiCorners = to_string(realTimeNormalization(numberOfShiTomasiKeyPoints,
i));

                //write SURF
                vectNumberOfKeyPoints.push_back(numberOfSURFFeatures);
                String numberOfKeyPointsSURF = to_string(realTimeNormalization(vectNumberOfKeyPoints, i));

                //write Harris
                numberOfHarrisCorners.push_back(numberOfHarrisCornersCounter);
                String strNumberOfHarrisCorners = to_string(realTimeNormalization(numberOfHarrisCorners, i));

                //if ready for OFA
                if(i > 10)
                {
                        opticalFlowAnalysisFarnebackNumbers.push_back(sumOpticalFlow);
                        strNumberOpticalFlowAnalysis =
to_string(realTimeNormalization(opticalFlowAnalysisFarnebackNumbers, i-11));
                        //compute FDOFA
                        finalScores.push_back(computeFinalScore(vectNumberOfKeyPoints, numberOfHarrisCorners,
numberOfShiTomasiKeyPoints, numberOfContours,
                                        opticalFlowAnalysisFarnebackNumbers, i));
                        strRating = to_string(realTimeNormalization(finalScores, i-11));
                }
                //if not enough data has been generated for optical flow
                else if(i > 0 && i <= 3)
                {

                        //creating text to display
                        strDisplay = "SURF Features: " + numberOfKeyPointsSURF + " Shi-Tomasi: " +
strNumberOfShiTomasiCorners + " Harris: "
                        + strNumberOfHarrisCorners + " Canny: " + strCanny + " Frame Number: " +
to_string(framesRead);

                        //creating black empty image
                        Mat pic = Mat::zeros(45,1910,CV_8UC3);

                        //adding text to image
                        putText(pic, strDisplay, cvPoint(30,30),CV_FONT_HERSHEY_SIMPLEX, 1.25, cvScalar(0,255,0),
1, CV_AA, false);

                        //displaying image
                        imshow("Stats", pic);

                }
```

```
                //gather real time statistics
                framesRead = (int) capture.get(CV_CAP_PROP_POS_FRAMES);
                framesTimeLeft = (capture.get(CV_CAP_PROP_POS_MSEC)) / 1000;

                //clocking end of run time
                clock_t tFinal = clock();

                //calculate time
                strActiveTimeDifference = (to_string(calculateFPS(tStart, tFinal))).substr(0, 4);

                //saving FPS values
                FPS.push_back(strActiveTimeDifference);

                //creating text to display
                strDisplay = "SURF: " + numberOfKeyPointsSURF + " Shi-Tomasi: " + strNumberOfShiTomasiCorners + "
Harris: "
                + strNumberOfHarrisCorners + + " Canny: " + strCanny + " FDOFA: " + strNumberOpticalFlowAnalysis
+  " Frame Number: " +
                to_string(framesRead) +  " Rating: " + strRating +  " FPS: " + strActiveTimeDifference;

                //creating black empty image
                Mat pic = Mat::zeros(45,1910,CV_8UC3);

                //adding text to image
                putText(pic, strDisplay, cvPoint(30,30),CV_FONT_HERSHEY_SIMPLEX, 1, cvScalar(0,255,0), 1, CV_AA,
false);

                //displaying image
                imshow("Stats", pic);

                //method to display frames
                //displayWindows(i);

                //read in current key press
                char c = cvWaitKey(33);

                //if escape key is pressed
                if(c==27)
                {
                        //reset key listener
                        c = cvWaitKey(33);

                        //display warning
                        cout << "Are you sure you want to exit?" << endl;

                        //if key is pressed again, in rapid succession
                        if(c==27)
                        {
                                //display exiting message
                                cout << "Exiting" << endl;

                                //normlize all ratings and metrics
                                normalizeRatings(vectNumberOfKeyPoints, numberOfShiTomasiKeyPoints,
numberOfHarrisCorners, numberOfContours, opticalFlowAnalysisFarnebackNumbers);

                                //save data to txt file
                                saveToTxtFile(FRAME_RATE, vectNumberOfKeyPoints, numberOfShiTomasiKeyPoints,
numberOfContours, numberOfHarrisCorners, opticalFlowAnalysisFarnebackNumbers,FPS, filename);

                                //compute total run time
                                computeRunTime(t1, clock(), (int) capture.get(CV_CAP_PROP_POS_FRAMES));

                                //close all windows
                                destroyWindows();
```

```
                        //delete entire vector
                        globalFrames.erase(globalFrames.begin(), globalFrames.end());

                        //report file finished writing
                        cout << "Finished writing file, Goodbye." << endl;

                        //exit program
                        return 0;
                }
        }

        //after keeping adequate buffer of 3 frames
        if(i > 3)
        {
                //deleting current frame from RAM
                delete frameToBeDisplayed;

                //replacing old frames with low RAM placeholder
                globalFrames.erase(globalFrames.begin() + (i - 3));
                globalFrames.insert(globalFrames.begin(), placeHolder);
        }

        //incrementing counter
        i++;
    }

    //delete entire vector
    globalFrames.erase(globalFrames.begin(), globalFrames.end());

    //normalize all ratings
    normalizeRatings(vectNumberOfKeyPoints, numberOfShiTomasiKeyPoints, numberOfHarrisCorners,
numberOfContours, opticalFlowAnalysisFarnebackNumbers);

    //save info in txt file
    saveToTxtFile(FRAME_RATE, vectNumberOfKeyPoints, numberOfShiTomasiKeyPoints, numberOfContours,
numberOfHarrisCorners, opticalFlowAnalysisFarnebackNumbers, FPS, filename);

    //compute run time
    computeRunTime(t1, clock(),(int) capture.get(CV_CAP_PROP_POS_FRAMES));

    //display finished, promt to close program
    cout << "Execution finished, file written, click to close window. " << endl;

    //wait for button press to proceed
    waitKey(0);

    //close all windows
    destroyWindows();

    //return code is finished and ran successfully
    return 0;
}
```