```
#ifndef HAAR_H_
#define HAAR_H_

#include "opencv2/opencv.hpp"
#include <math.h>
#include <stdio.h>
#include <vector>   // for splitted image

#define SQRT_2  std::sqrt(2.0f)

using namespace cv;
using namespace std;

/**
 ** http://stackoverflow.com/questions/20071854/wavelet-transform-in-opencv
 **/

// Filter type
#define NONE 0      // no filter
#define HARD 1      // hard shrinkage
#define SOFT 2      // soft shrinkage
#define GARROT 3    // Garrot filter

namespace zs {
typedef enum
{
    HAAR_FITER_NONE,
    HAAR_FILTER_HARD,
    HAAR_FILTER_SOFT,
    HAAR_FILTER_GARROT
}_Haar_Filter_Type;
}

// window
#define __WINDOW_IMAGE__ "image"
#define __WINDOW_VIDEO__ "video"
#define __WINDOW_FILTERED__ "filtered"
#define __WINDOW_HAAR__ "haar"
#define __WINDOW_HAAR_INV__ "haar-inv"

//--------------------------------
// signum
//--------------------------------
float sgn(float x) {
    float res=0;
    if(x==0) { res=0.f; }
    if(x>0) { res=1.f; }
    if(x<0) { res=-1.f; }
    return res;
}

//--------------------------------
// Soft shrinkage
//--------------------------------
float soft_shrink(float d,float T) {
    float res;
    if(fabs(d)>T) { res=sgn(d)*(fabs(d)-T); }
    else { res=0; }
    return res;
}

//--------------------------------
// Hard shrinkage
//--------------------------------
float hard_shrink(float d,float T) {
    float res;
    if(fabs(d) > T) { res=d; } else { res=0; }
    return res;
}

//--------------------------------
```

```
// Garrot shrinkage
//-------------------------------
float Garrot_shrink(float d,float T) {
    float res;
    if(fabs(d) > T) { res=d-((T*T)/d); }
    else { res=0; }
    return res;
}

//-------------------------------
// Wavelet transform
//-------------------------------
static void cvHaarWavelet(Mat &src,Mat &dst,int NIter) {
    float   c,
        dh,     // horizontal details
        dv,     // vertical details
        dd;     // global details
    assert( src.type() == CV_32FC1 );
    assert( dst.type() == CV_32FC1 );
    int width = src.cols;
    int height = src.rows;

    for (int k=0;k<NIter;k++) // levels
    {
        for (int y=0; y < ( height >> (k+1) ); y++)
        {
            for (int x=0; x < ( width >> (k+1) ) ;x++)
            {
                // first branch is per column operation, the rest is row-wise operation
                c=  (   (src.at<float>(2*y,2*x) + src.at<float>(2*y,2*x+1)) +
                        (src.at<float>(2*y+1,2*x) + src.at<float>(2*y+1,2*x+1))
                    )   *(1/std::sqrt(2.0f));//0.5f;
                dst.at<float>(y,x)=c;

                dh=(    (src.at<float>(2*y,2*x) + src.at<float>(2*y+1,2*x)) -
                        (src.at<float>(2*y,2*x+1) - src.at<float>(2*y+1,2*x+1))
                   )    *(1/std::sqrt(2.0f));//0.5f;
                dst.at<float>(y,x+(width>>(k+1)))=dh;

                dv=( (src.at<float>(2*y,2*x) + src.at<float>(2*y,2*x+1)) - (src.at<float>(2*y+1,2*x) - ↙
    src.at<float>(2*y+1,2*x+1)) )* (1/std::sqrt(2.0f));//0.5f;
                dst.at<float>(y+(height>>(k+1)),x)=dv;

                dd=( (src.at<float>(2*y,2*x) - src.at<float>(2*y,2*x+1)) - (src.at<float>(2*y+1,2*x) + ↙
    src.at<float>(2*y+1,2*x+1)) )* (1/std::sqrt(2.0f));//0.5f;
                dst.at<float>(y+(height>>(k+1)),x+(width>>(k+1)))=dd;
            }
        }
        dst.copyTo(src);    // why this guy needs this again???
    }
}

//-------------------------------
//Inverse wavelet transform
//-------------------------------
static void cvInvHaarWavelet(Mat &src,Mat &dst,int NIter, int SHRINKAGE_TYPE=zs::HAAR_FITER_NONE, float ↙
    SHRINKAGE_T=50) {
    float c,dh,dv,dd;
    assert( src.type() == CV_32FC1 );
    assert( dst.type() == CV_32FC1 );
    int width = src.cols;
    int height = src.rows;
    //-------------------------------
    // NIter - number of iterations
    //-------------------------------
    for (int k=NIter;k>0;k--)
    {
        for (int y=0;y<(height>>k);y++)
        {
            for (int x=0; x<(width>>k);x++)
            {
```

```
                c=src.at<float>(y, x);
                dh=src.at<float>(y, x + ( width >> k ));
                dv=src.at<float>(y + ( height >> k ), x);
                dd=src.at<float>(y + ( height >> k ), x + ( width >> k ));

                // (shrinkage)
                switch(SHRINKAGE_TYPE)
                {
                case HARD:
                    dh=hard_shrink(dh,SHRINKAGE_T);
                    dv=hard_shrink(dv,SHRINKAGE_T);
                    dd=hard_shrink(dd,SHRINKAGE_T);
                    break;
                case SOFT:
                    dh=soft_shrink(dh,SHRINKAGE_T);
                    dv=soft_shrink(dv,SHRINKAGE_T);
                    dd=soft_shrink(dd,SHRINKAGE_T);
                case GARROT:
                    dh=Garrot_shrink(dh,SHRINKAGE_T);
                    dv=Garrot_shrink(dv,SHRINKAGE_T);
                    dd=Garrot_shrink(dd,SHRINKAGE_T);
                    break;
                }

                //--------------------
                dst.at<float>(y*2,x*2)=0.5f*(c+dh+dv+dd);
                dst.at<float>(y*2,x*2+1)=0.5f*(c-dh+dv-dd);
                dst.at<float>(y*2+1,x*2)=0.5f*(c+dh-dv-dd);
                dst.at<float>(y*2+1,x*2+1)=0.5f*(c-dh-dv+dd);
            }
        }
        Mat C=src(Rect(0,0,width>>(k-1),height>>(k-1)));
        Mat D=dst(Rect(0,0,width>>(k-1),height>>(k-1)));
        D.copyTo(C);
    }
}

//  --------------------------------
//  main process occurs here
//  --------------------------------
int process(VideoCapture& capture)
{
    int n = 0;
    const int NIter=4;
    char filename[200];
    string window_name = "video | q or esc to quit";
    cout << "press space to save a picture. q or esc to quit" << endl;
    namedWindow(window_name, CV_WINDOW_KEEPRATIO); //resizable window;
    Mat frame;
    capture >> frame;

    Mat GrayFrame=Mat(frame.rows, frame.cols, CV_8UC1);
    Mat Src=Mat(frame.rows, frame.cols, CV_32FC1);
    Mat Dst=Mat(frame.rows, frame.cols, CV_32FC1);
    Mat Temp=Mat(frame.rows, frame.cols, CV_32FC1);
    Mat Filtered=Mat(frame.rows, frame.cols, CV_32FC1);
    while (1)
    {
        Dst=0;
        capture >> frame;
        if (frame.empty()) continue;
        cvtColor(frame, GrayFrame, CV_BGR2GRAY);
        GrayFrame.convertTo(Src,CV_32FC1);
        cvHaarWavelet(Src,Dst,NIter);

        Dst.copyTo(Temp);

        cvInvHaarWavelet(Temp,Filtered,NIter,GARROT,30);

        cv::imshow(window_name, frame);
```

```
        double M=0,m=0;
        //--------------------------------------------------
        // Normalization to 0-1 range (for visualization)
        //--------------------------------------------------
        cv::minMaxLoc(Dst,&m,&M);
        if((M-m)>0) {Dst=Dst*(1.0/(M-m))-m/(M-m);}
        cv::imshow("Coeff", Dst);

        cv::minMaxLoc(Filtered,&m,&M);
        if((M-m)>0) {Filtered=Filtered*(1.0/(M-m))-m/(M-m);}
        cv::imshow("Filtered", Filtered);

        char key = (char)waitKey(5);
        switch (key)
        {
        case 'q':
        case 'Q':
        case 27: //escape key
            return 0;
        case ' ': //Save an image
            sprintf(filename,"filename%.3d.jpg",n++);
            imwrite(filename,frame);
            cout << "Saved " << filename << endl;
            break;
        default:
            break;
        }
    }
    return 0;
}

// Haar transformation class
namespace zs {
class HAAR_TRANSFORM {
public:
    HAAR_TRANSFORM(const cv::Mat& src, cv::Mat& dst, int filterType=zs::HAAR_FITER_NONE) : d_(&dst), i_ ↙
    (src) {
        // number of iteration by default is 2
        n_ = 2;

        // split image based on its channel
        cv::split(i_, splittedImage_);

        // for haar conversion, needs to rescale the data value to 32 bit floating point
        splitImgIterator = splittedImage_.begin();
        for(splitImgIterator = splittedImage_.begin();
            splitImgIterator != splittedImage_.end();
            ++splitImgIterator) {
                splitImgIterator->convertTo(*splitImgIterator, CV_32FC1);
        }
    }

    inline void SetNumberOfIteration(int n) {
        n_ = n;
    }

    void Do() {
        for(splitImgIterator = splittedImage_.begin() ; splitImgIterator != splittedImage_.end(); ++    ↙
    splitImgIterator) {
            DoHaarTransorm(splitImgIterator);
        }
    }

protected:
    inline void DoHaarTransorm(/*cv::Mat**/ std::vector<cv::Mat>::iterator singleChannelImg) {
        float c,
            dh,
            dv,
            dd;
        assert(singleChannelImg->type() == CV_32FC1); // check if its 32 floating point or not
        int w = singleChannelImg->cols;
```

```
        int h = singleChannelImg->rows;

        cv::Mat buff(singleChannelImg->rows, singleChannelImg->cols, CV_32FC1,  cv::Scalar::all(0.0));

        // begin

        for(int k = 0; k < n_; ++k) // n iter
        {
            // now access individual column and rows
            for (int y = 0; y < (h >> (k+1)); ++y)  // rows
            {
                for (int x = 0; x < (w >> (k+1)); ++x)  // cols
                {
                    c = ( (singleChannelImg->at<float>(2*y,2*x) + singleChannelImg->at<float>(2*y,2*x+1)↙
     ) +
                          (singleChannelImg->at<float>(2*y+1,2*x) + singleChannelImg->at<float>(2*y+1,2*↙
    x+1))
                        )*(1.0f/SQRT_2);     // better use constant, no need to re calc sqrt(2)

                    dh = ( (singleChannelImg->at<float>(2*y,2*x) + singleChannelImg->at<float>(2*y+1, 2*↙
    x)) -
                           (singleChannelImg->at<float>(2*y,2*x+1)-singleChannelImg->at<float>(2*y+1,2*x↙
    +1))
                         )*(1/SQRT_2);

                    dv = (  (singleChannelImg->at<float>(2*y,2*x) + singleChannelImg->at<float>(2*y,2*x+↙
    1)) -
                           (singleChannelImg->at<float>(2*y+1,2*x)-singleChannelImg->at<float>(2*y+1,2*↙
    x+1))
                         )*(1/SQRT_2);

                    dd = (  (singleChannelImg->at<float>(2*y,2*x)-singleChannelImg->at<float>(2*y,2*x+  ↙
    1)) -
                           (singleChannelImg->at<float>(2*y+1,2*x)+singleChannelImg->at<float>(2*y+1,2*↙
    x+1))
                         )  *(1.0f/SQRT_2);

                    buff.at<float>(y,x) = c;
                    buff.at<float>(y,x+(w>>(k+1))) = dh;
                    buff.at<float>(y+(h>>(k+1)),x) = dv;
                    buff.at<float>(y+(h>>(k+1)),x+(w>>(k+1))) = dd;
                } //  x
            } // y
            buff.copyTo(*singleChannelImg);
        }   // k
    }
private:
    cv::Mat *d_, i_;
    // splitted image
    std::vector<cv::Mat> splittedImage_;    // single channel image
    std::vector<cv::Mat>::iterator splitImgIterator;

    // number of iteration
    int n_;
};
}

#endif // !HAAR_H_
```