```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
using System.Threading;

namespace GAWavelet
{
    /// <summary>
    /// This class contains methods that take as input and image and some settings and use
    /// them to transform that image and display the results of the transformation.
    /// The transformation may be a straight wavelet transform or one that is found via
    /// genetic algorithm
    /// </summary>
    public class Transform
    {
        declare variables

        public Transform()
        {
            transformComplete = false;
        }

        /// <summary>
        /// this method performs the wavelet transform (via Genetic Algorithm if specified
     in
        /// settings) and spawns a results dialog upon completion
        /// </summary>
        /// <param name="imageControl">
        /// a refference to the set of all images loaded
        /// </param>
        /// <param name="ts">
        /// a refference to the convenience object holding the bundled transform settings
        /// </param>
        /// <param name="selectedImage">
        /// which image the transform is to be applied to
        /// </param>
        public void PerformTransform(ref ImageControl[] imageControl, ref
    TransformSettings ts, int selectedImage)
        {
            //Thread MyThread = new Thread(new ThreadStart (Transforming));
            //MyThread.Start();

            Init variables based on settings
            imageControl[selectedImage].calcOddYUV();                    // convert image
    to odd YUV
            double [] Ysrc = imageControl[selectedImage].getOddY();      // load in Y U V
    arrays...
            double [] Usrc = imageControl[selectedImage].getOddU();
            double [] Vsrc = imageControl[selectedImage].getOddV();
            double [] Ydst = new double[width*height];                   // init
    destination arrays with copies of source
            double [] Udst = new double[width*height];
            double [] Vdst = new double[width*height];
            Ysrc.CopyTo(Ydst,0);
            Usrc.CopyTo(Udst,0);
            Vsrc.CopyTo(Vdst,0);
            public_progress = 20;
            Forward Transform (calls)

            public_progress = 30;

            Quantize
            public_progress = 40;
            EncodeFrame (to calculate compressed file size)
```

```
        public_progress = 60;
        Dequantize
        public_progress = 70;
        Inverse Transform (calls)
        public_progress = 90;
        Gentic Agorithm
        bestSetFound = new CoefficientSet();
        bestSetFound.hlen = hlen;
        bestSetFound.glen = glen;
        bestSetFound.h1 = (double[])best_h1.Clone();
        bestSetFound.g1 = (double[])best_g1.Clone();
        bestSetFound.h2 = (double[])best_h2.Clone();
        bestSetFound.g2 = (double[])best_g2.Clone();

        imageControl[selectedImage].setOddYUVOrign(ref Ysrc,ref Usrc,ref Vsrc);
        imageControl[selectedImage].setOddYUV(ref bestY,ref Udst,ref Vdst);
        completeMSE = imageControl[selectedImage].getMSE();
        filesizeC = (int)winning_Y_Length + original_U_Comp_Length +
original_V_Comp_Length;
        transformComplete = true;
        // show the results dialog
    }



    // used to perturb initial population.
    static double GetRandomNoiseFactor(Random r)
    {
        double noise;
        noise =  r.Next() % 1000;
        if (noise <950)
        {
            noise = (1 + ((475 - noise) / 2550));
            return noise;}
        else
            return -1.0;
    }

    // used to mutate population. returns small random value used to adjust
    //
    static double GetMutationFactor(Random r)
    {
        double noise;
        noise = r.Next() % 1000;
        if (noise <950)
        {
            noise = Math.Sqrt(1 + ((475 - noise) / 2275));
            return noise;}
        else
            return -1.0;
    }

    // perform forward wavelet transform with seed coefficients
    public void ForwardTransform(ref double[] src, int width, int height, int
mrlevels)
    {
        double[] temp = new double[height];
        int lengthw = width;
        int lengthh = height;

        for(int k=0;k<mrlevels;k++)
        {
            if((lengthw < hlen)||(lengthh < hlen)||(lengthw < glen)||(lengthh < glen))
            {
                mrlevels = k;
                break;
            }
            // Horizontal
```

```
            for(int y=0;y<lengthh;y++)
                forward(ref src,lengthw,h1,g1,hlen,glen,(y*width));
            // vertical
            for(int x=0;x<lengthw;x++)
            {
                for(int y=0;y<lengthh;y++) temp[y] = src[x+y*width];
                forward(ref temp,lengthh,h1,g1,hlen,glen, 0);
                for(int y=0;y<lengthh;y++) src[x+y*width] = temp[y];
            }
            lengthw = lengthw/2 + lengthw%2;
            lengthh = lengthh/2 + lengthh%2;
        }
    }

    // perform reverse wavelet transform with seed coefficients
    public void InverseTransform (ref double[] src, int width, int height, int
mrlevels)
    {
        int x, y, k;
        double[] temp;
        int lengthw, lengthh;

        temp = new double[height];

        for(k = (mrlevels-1);k>=0;k--)
        {
            lengthw = width>>k;
            lengthh = height>>k;

            if(width%((int)Math.Pow(2,k)) != 0) lengthw++;
            if(height%((int)Math.Pow(2,k)) != 0) lengthh++;

            // vertical
            for(x=0;x<lengthw;x++)
            {
                for(y=0;y<lengthh;y++) temp[y] = src[x+y*width];
                inverse(ref temp,lengthh,h2,g2,glen,hlen,0);
                for(y=0;y<lengthh;y++) src[x+y*width] = temp[y];
            }
            // horizontal
            for(y=0;y<lengthh;y++) inverse(ref src,lengthw,h2,g2,glen,hlen,y*width);

        }
    }

    // helper function for forward wavelet transform
    static void forward(ref double[] X, int N, double[] h, double[] g, int hlen, int
glen, int offset)
    {

        int i, m, len;
        double[] H, G;
        int odd, index;
        double oddvalue = 0;
        int hoffset, goffset;

        odd = N%2;
        if(odd>0)
        {
            len = N-1;
            oddvalue = X[N-1+offset];
        }
        else
            len = N;

        hoffset = (hlen/2)-1;
        goffset = (glen-1)/2;
```

```
        H = new double[len];
        G = new double[len];

        for(i=0;i<len;i++)
        {
            H[i] = 0;
            G[i] = 0;
            for(m=0;m<hlen;m++)
            {
                index = i+m-hoffset;
                if(index < 0)
                    index = index + len;
                else if(index > len-1)
                    index = index - len;
                H[i] = H[i]+h[m]*X[index+offset];
            }
            for(m=0;m<glen;m++)
            {
                index = i+m-goffset;
                if(index < 0)
                    index = index + len;
                else if(index > len-1)
                    index = index - len;
                G[i] = G[i]+g[m]*X[index+offset];
            }
        }
        for(i=0;i<len/2;i++)
        {
            X[i+offset] = H[i*2];
            X[i+(int)(len/2)+odd+offset] = G[i*2];
        }
        if(odd>0) X[(len/2)+offset] = oddvalue;
    }

    // helper function for inverse wavelet transform
    static void inverse(ref double[] X, int N, double[] h, double[] g, int hlen, int glen, int offset)
    {
        int i, m, len;
        double[] H;
        double[] G;
        double[] upH;
        double[] upG;
        int odd, index;
        double oddvalue = 0;
        int hoffset, goffset;

        odd = N%2;
        if(odd>0)
        {
            len = N-1;
            oddvalue = X[len/2];
        }
        else
            len = N;

        hoffset = (hlen+1)>>1;
        goffset = glen>>1;

        H = new double[len];
        G = new double[len];
        upH = new double[len];
        upG = new double[len];

        // Upsample both high and low components;
        for(i=0;i<len;i+=2)
        {
            upH[i] = X[i/2 + offset];
```

```
            upG[i] = X[i/2+odd+len/2 + offset];
            upH[i+1] = 0;
            upG[i+1] = 0;
        }

        for(i=0;i<len;i++)
        {
            H[i] = 0;
            G[i] = 0;
            for(m=0;m<hlen;m++)
            {
                index = i+m-hoffset;
                if(index < 0)
                    index = index + len;
                else if(index > len-1)
                    index = index - len;
                H[i] = H[i]+h[m]*upH[index];
            }
            for(m=0;m<glen;m++)
            {
                index = i+m-goffset;
                if(index < 0)
                    index = index + len;
                else if(index > len-1)
                    index = index - len;
                G[i] = G[i]+g[m]*upG[index];
            }
        }
        for(i=0;i<len;i++)
        {
            X[i+offset] = H[i]+G[i];
        }
        if(odd>0) X[N-1+offset] = oddvalue;
    }


    // Encode the background in the VO compression mode
    char bitstack;
    short bitcount, bitpos;
    int coeffval;
    int zerorun;                        //unsigned?
    int buflength;
    char[] buf;
    int bufPointer;
    const short plus = 3;
    const short minus = 2;
    const short one = 1;
    const short zero = 0;

    char[] EncodeFrame(ref double [] GOF, int width, int height, int level2D, ref int ↙
codelength)
    {
        char[] retbuf;
        int startw,endw,starth,endh;
        int w,h,k;

        // Allocate temporary buffer for the encoded frame
        buf = new char[width*height*8];
        bufPointer = 0;

        // initialize various parameters
        buflength = 0;
        bitstack = (char)0;
        bitcount = 0;
        zerorun = 0;

        // the end vertical and horizontal
        // positions of the subband
```

```
endw = width;
endh = height;
for(k=0;k<level2D;k++)
{
    // break if the subband is 2 pixels
    // or less in either direction
    if((endw < 2) || (endh < 2))
        break;

    // Find starting end ending values of the subbands
    startw = width>>(k+1);
    if(width%((int)Math.Pow(2,k+1)) != 0)
        startw++;
    starth = height>>(k+1);
    if(height%((int)Math.Pow(2,k+1)) != 0)
        starth++;

    // Encode the Subbands --

    //HH subband
    for(h=starth;h<endh;h++)
        for(w=startw;w<endw;w++)
            writevalue(GOF[w+width*h]);
    //HL subband
    for(w=startw;w<endw;w++)
        for(h=0;h<starth;h++)
            writevalue(GOF[w+width*h]);
    //LH subband
    for(h=starth;h<endh;h++)
        for(w=0;w<startw;w++)
            writevalue(GOF[w+width*h]);

    // keep track of the LL band
    endw = startw;
    endh = starth;
}

// Encode the LL band
for(h=0;h<endh;h++)
    for(w=0;w<endw;w++)
        writevalue(GOF[w+width*h]);

// Encode the remaining run length
if(zerorun != 0)
{
    writestack(zerorun, true);
    zerorun = 0;
}
// Write the remaining bit-stack
if(bitstack != 0)
{

    buf[bufPointer] = bitstack;
    bufPointer+=1;
    buflength+=1;
    bitstack = (char)0;
    bitcount = 0;
}

// return buffer pointer
//buf = buf - buflength;
// generate a new buffer with length of the encoded video

// copy the encoded data
//MessageBox.Show(""+buflength);
//retbuf = buf.ToString().ToCharArray(0,buflength-1);
// return buffer length
```

```
    codelength = buflength;
    // free temporary buffer


    return buf;
}

// Stack-run Encoding
// Write a wavelet coefficient value
// into the stack-run encoder
void writevalue(double value)
{
    // If there is no zero run, and
    // the next coefficient is not zero,
    // just write the coefficient
    if((zerorun == 0) && (value != 0))
    {
        writestack(value, false);
    }
        // If there is a zero run, and
        // the next coefficient is not zero,
        // then we have to write the run value
        // and the coefficient
    else if(value != 0)
    {
        writestack(zerorun, true);
        zerorun = 0;
        writestack(value, false);
    }
        // If the value is zero, then we just
        // add to the zero run
    else //if(value == 0)
        zerorun++;
}

void writestack(double dvalue, bool writerun)
{
    short bitval;
    int value;
    bool negative;
    bool writemsb;

    // find if a negative coefficient has
    // been written
    if((dvalue < 0) && (writerun == false))
    {
        value = -(int)dvalue;
        // we must add one to the coefficent value
        // so coefficients cannot be confused with
        // run values
        value++;
        negative = true;
    }
        // find if a positive coefficent has
        // been written
    else if(writerun == false)
    {
        // we must add one to the coefficent value
        // so coefficients cannot be confused with
        // run values
        value = (int)dvalue+1;
        negative = false;
    }
    else
    {
        value = (int)dvalue;
        negative = false;
    }
```

```
// we assume first a coefficient value, or
// a small run value.  Large run values and coefficient values do
// not need to have the msb written to the file
writemsb = false;
while(value != 0)
{
    // if the bit-stack is full (8 bits)
    // it needs to be written to the encoded buffer
    if(bitcount == 4)
    {
        buf[bufPointer] = bitstack;
        bufPointer+=1;
        buflength+=1;
        bitstack = (char)0;
        bitcount = 0;
    }
    // extract the bit value
    bitval = (short)(0x0001&value);
    value = value>>1;

    if(bitval == 0)
    {
        // if the bitvalue is zero, then this must
        // be either a large run value or a coefficient value.
        // either way, we do not have to write the msb
        writemsb = false;
        // Write a run symbol for 0
        if(writerun == false)
        {
            bitstack = (char)((zero<<(2*bitcount))|bitstack);
        }
            // write a coefficient symbol for 0
        else
        {
            bitstack = (char)((minus<<(2*bitcount))|bitstack);
        }
        bitcount++;
    }
    else
    {
        // Write a run symbol for 1
        if(writerun == false)
        {
            bitstack = (char)((one<<(2*bitcount))|bitstack);
            writemsb = false;
        }
            // write a coefficient symbol for 1
        else
        {
            bitstack = (char)((plus<<(2*bitcount))|bitstack);
        }
        bitcount++;
    }

    // if the bit-stack is full (8 bits)
    // it needs to be written to the encoded buffer
    if(bitcount == 4)
    {
        buf[bufPointer] = bitstack;
        bufPointer+=1;
        buflength+=1;
        bitstack = (char)0;
        bitcount = 0;
    }

    // if the last bit (msb) is left, and we don't have
    // to write the msb, then break
```

```
                    if((value == 1)&&(negative == true)&&(writemsb==false))
                    {
                        // write the sign of the coefficient value
                        if(writerun == false)
                        {
                            bitstack = (char)((minus<<(2*bitcount))|bitstack);
                            bitcount++;
                        }
                        value = 0;
                        break;
                    }
                    else if((value == 1)&&(negative == false)&&(writemsb==false))
                    {
                        // write the sign of the coefficient value
                        if(writerun == false)
                        {
                            bitstack = (char)((plus<<(2*bitcount))|bitstack);
                            bitcount++;
                        }
                        value = 0;
                        break;
                    }
                        // break if the value has been completely written
                    else if(value == 0)
                    {
                        break;
                    }
                }
            }
        }
    }
```