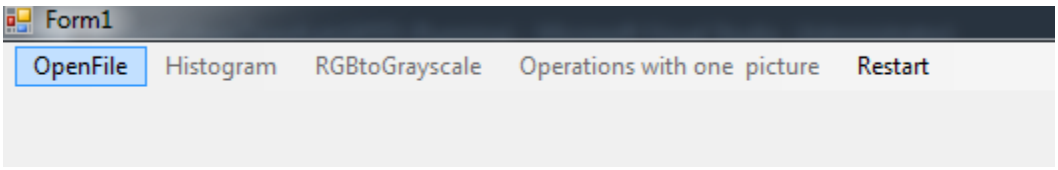
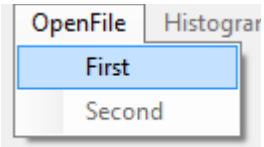
Dokumentacja projektu z C#

Wykonal:Taras Kuts 16555

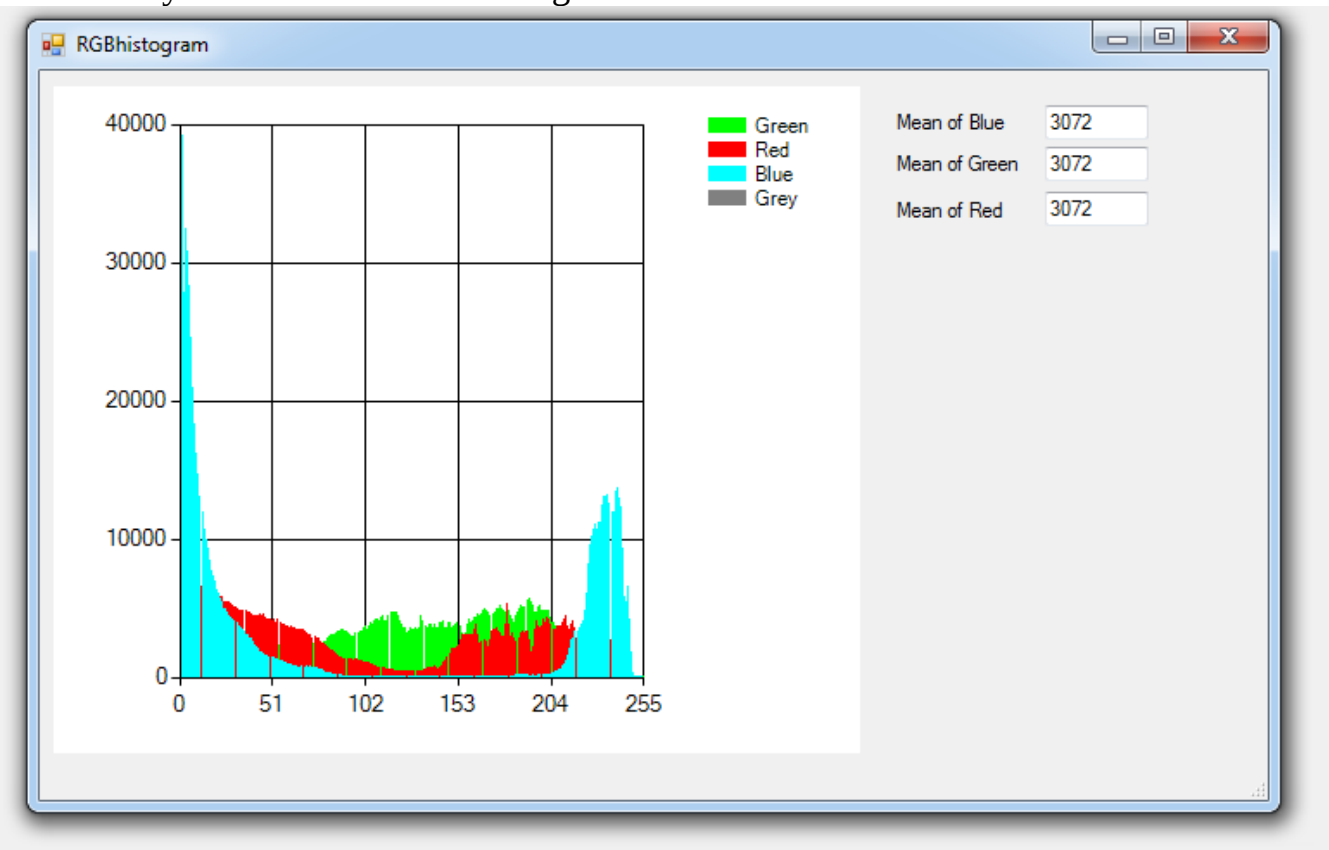
W projekcie jest realizowane następne:
Histogram RGB/Grayscale,przeksztalcenia w grayscale,nakladanie róznych masek około 18 oraz możliwość nakladania wlasnej,scalowanie,nakladanie pikselow brzygowych, rozciągnięcia histogramu z wartosciami Min i Max,Treshhold z wartościami Min i Max,redukcjia z parametrami qi,pi,Treshhold z jedną wartościu,dilacja,erozja,opening,closing,negacja,compression za algorytmama Read,code Huffmana i Compression with parametres,equalizing and stratching,mediana różne rozmiary z opcja nakladania od razu pikselej brzygowych.Też operacje na dwoch obrazach taki jak ADD,DIFF,SUB,OR,XOR,AND.Dla nakladanie dilacji,erozji,opening i closing jest wykorzystana biblioteka Aforge.
Przykladowe działanie programy:
Przy starcie programy



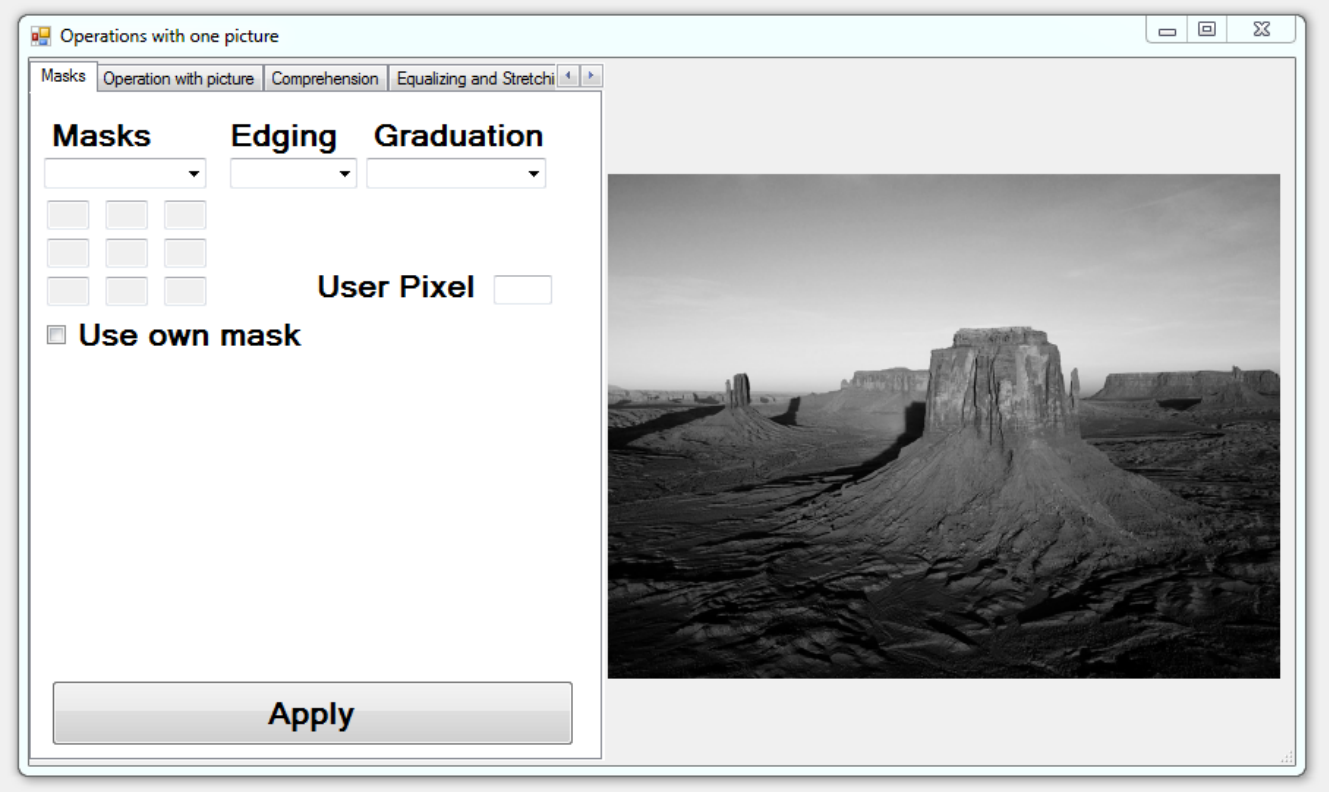
Mozemy tylko nacisnąć OpenFile lub Restart.First to jest pierwszyj obraz



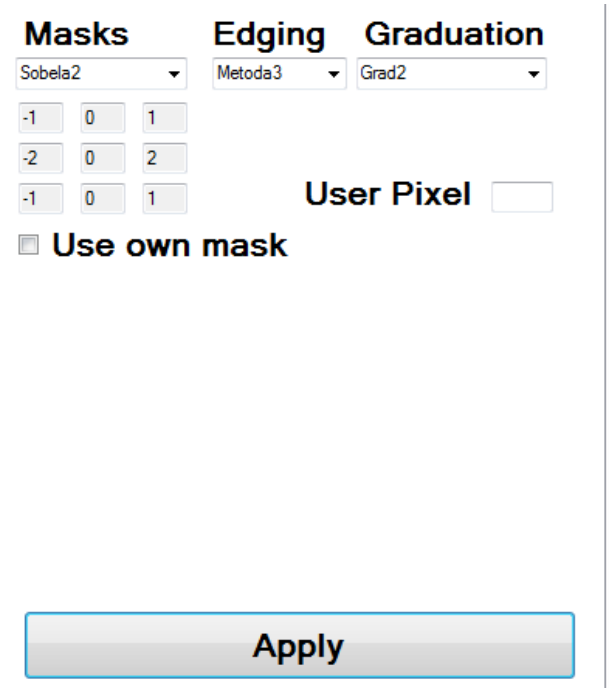
Możemy ten obraz zwiększyć lub zmniejszyć.Dalej widzimy że możemy już używać RGBtoGraysclafe-Picture1 lub Histogram-RGB



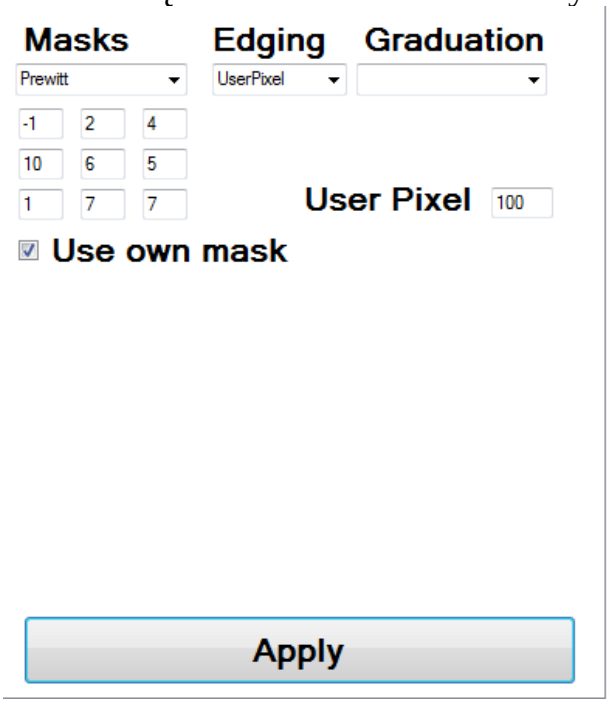
Po grayscale możemy już działać z obrazem.



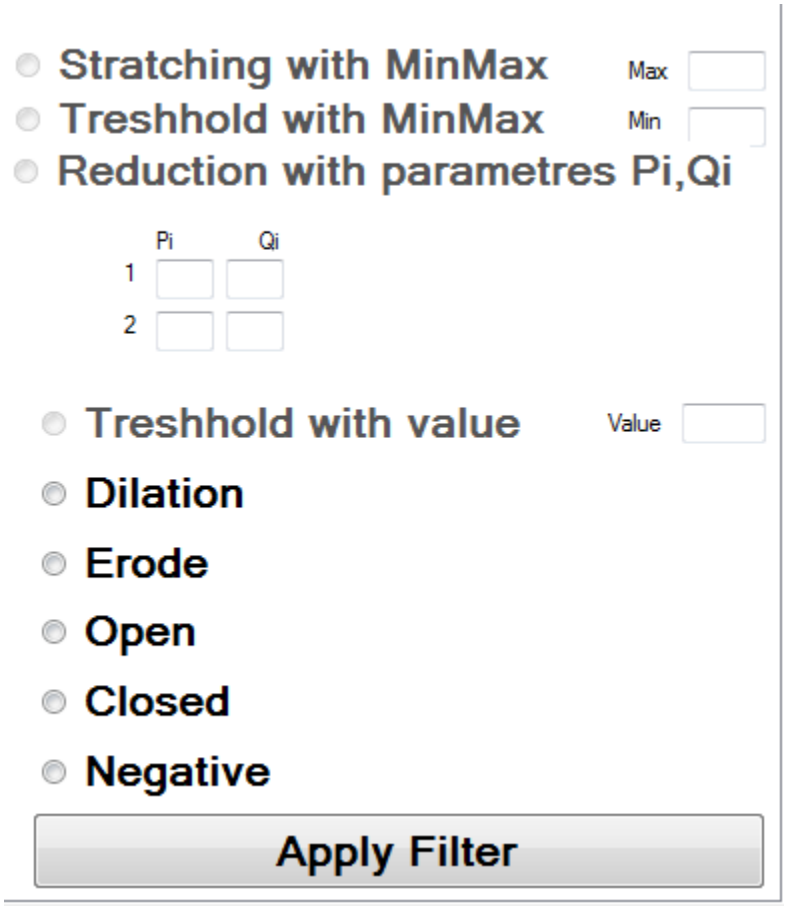
Odrazu mamy przeksztalcony obraz w grayscali jake po nacisniecu na go mozemy Zmniejszyc/Zblizac lub Save.

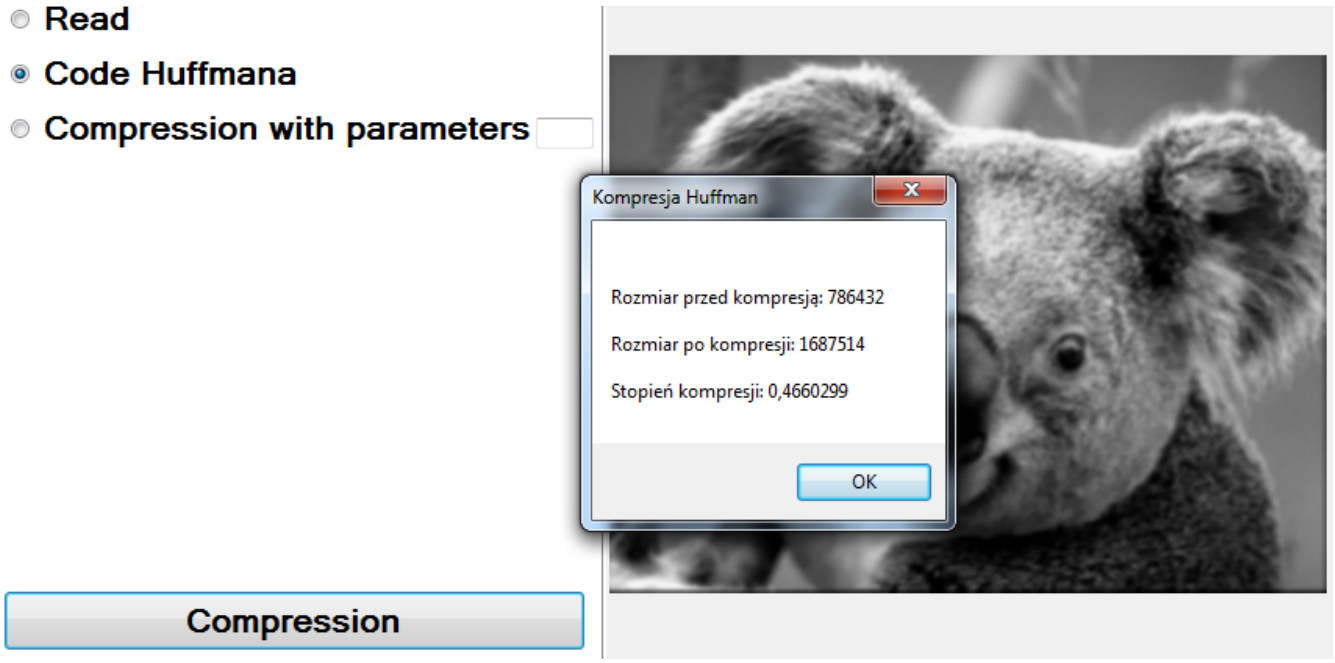


Po naciśnięciu Use own mask możemy wpisywać wlasną maske.

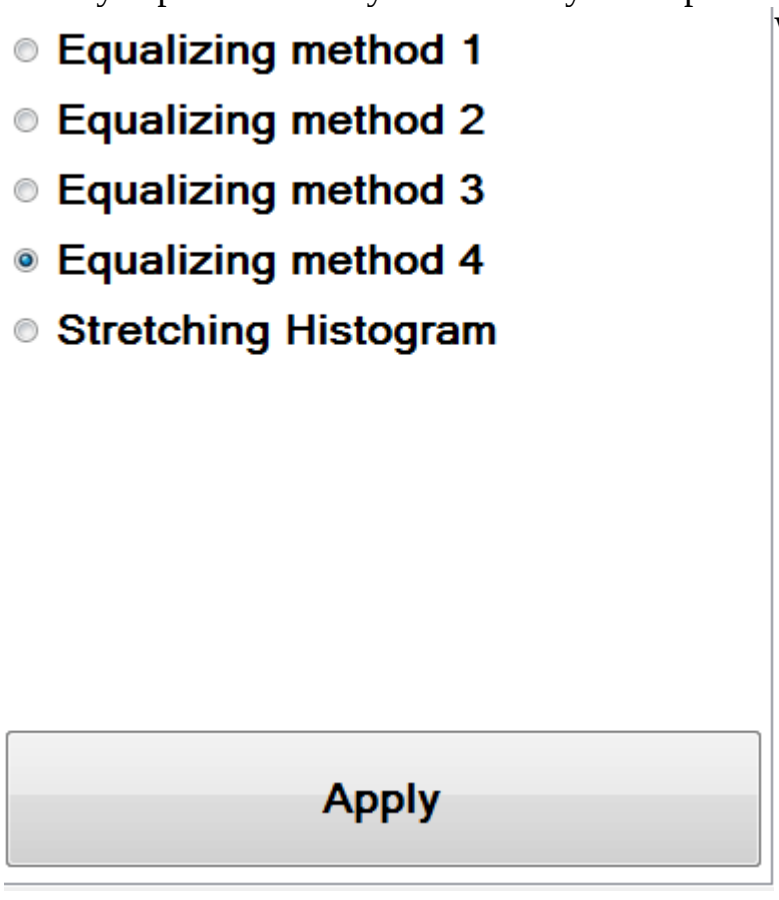


Stratching with MinMax,Treshhold with MinMax,Reduction with parametres Pi,Qi i Treshhold with value mozemy wybrac odpowiednie kiedy wpiszemy wartosci w kazde pole odpowiednie.

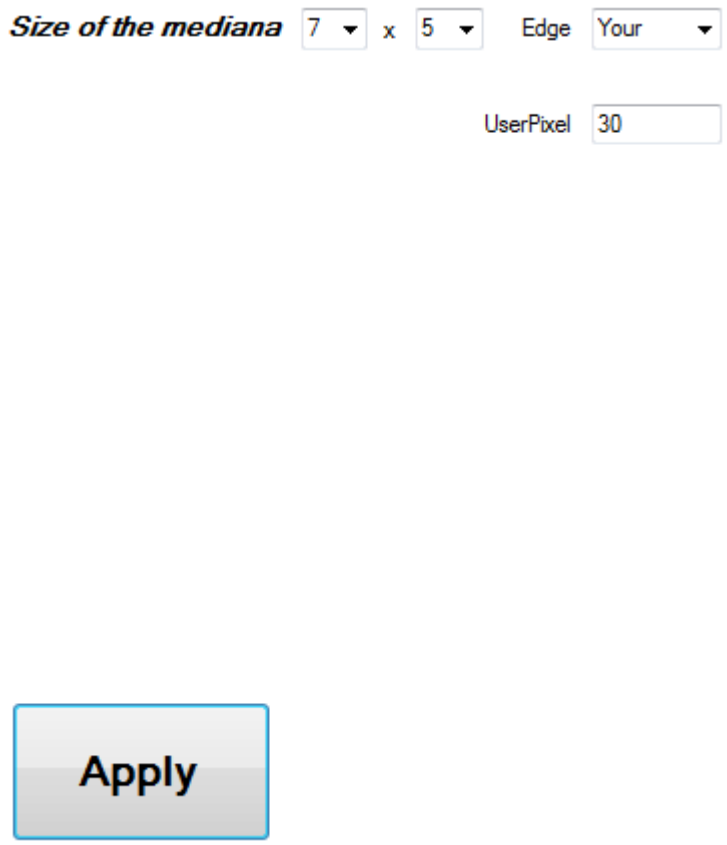




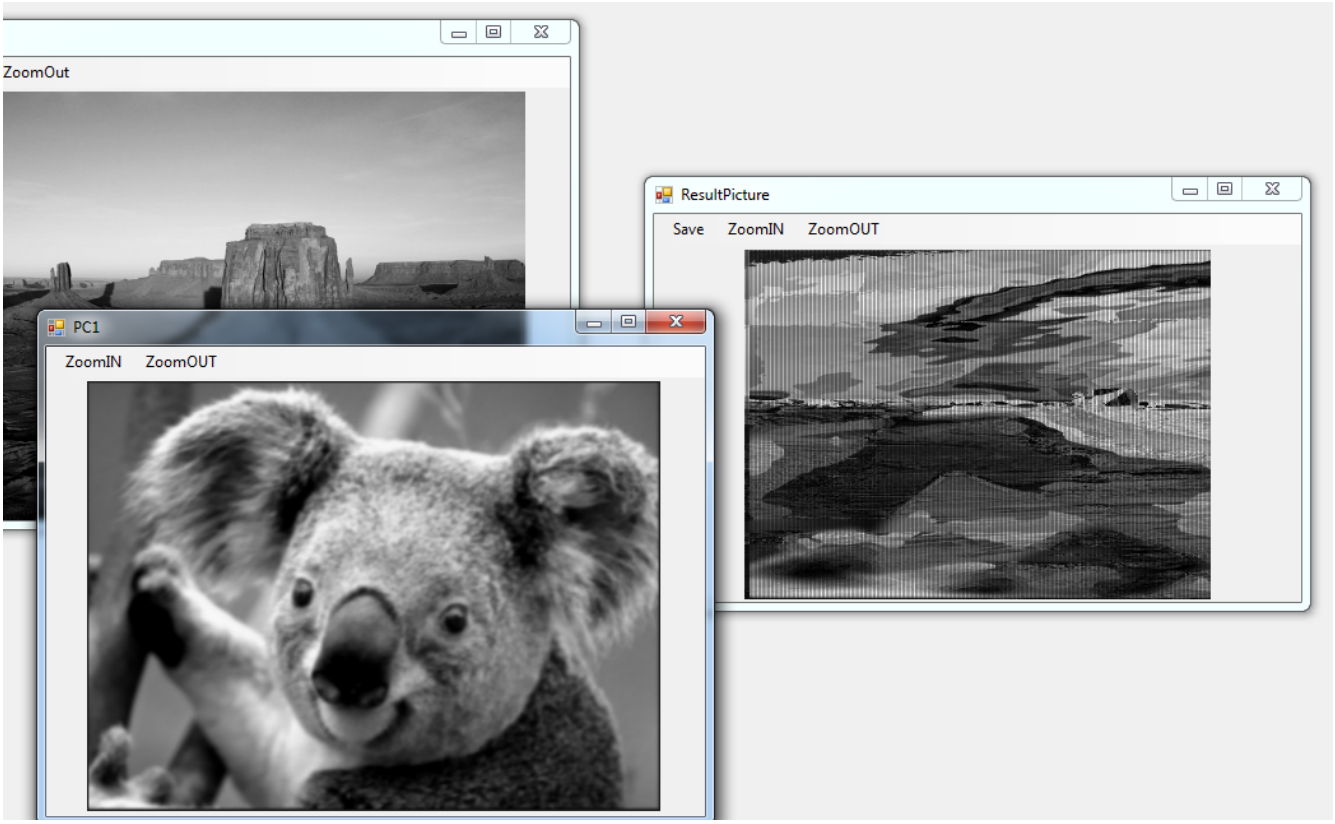
Mozemy zapisac i zobaczyc ze obraz był skompersowany.



Wybieramy naciskamy „Apply”



Zeby można było nacisnac operation with two pictures trzeba utworzyc jeszcze jedna i zrobic w graysclae wtydy jest już dostepna mozliwosc dzialania z ta opcja.



Wynik peracji XOR , wynik możemy zapisać.
Kod:Class ImageManipulation

```
        using AForge.Imaging.Filters;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Csharp
{
    enum Graduation { first, second, third };
    enum maskSize { three = 0, five = 1, seven = 2 };
    enum edgeMethods { first, second, third, mine };
    enum masksAplied { onNine, onTen, onSixteen, cyfrowaLaplas, laplasA, laplasB, laplasC, laplasD, edgeOne, edgeTwo,
edgeThree, Roberts, Roberts2, Prewitt, Prewitt2, Uniwersal1, Uniwersal }
    enum equalizationsMetods { metoda1, metoda2, metoda3, metoda4 };
    enum ArithmeticOperations { ADD, SUB, DIFF, OR, AND, XOR };
    enum Histogram { HistogramRGB, Histogram };
    class ImageManipulation
    {
        public Bitmap imgBitmapCOPY { set; get; }
        public Bitmap imgBitmap2 { set; get; }
        public Bitmap imgBitMap { set; get; }
        public List<long[]> ListOfHistogramGrey = new List<long[]>();
        public List<long[]> ListOfHistogramRGB = new List<long[]>();
        long[] histogramBlue = new long[256];
        long[] histogramRed = new long[256];
        long[] histogramGreen = new long[256];
        long[] histogramGray = new long[256];
        int min = 255;
```

```
        int max = 0;
        static int levels = 256;
        public double histogramAVG { get; set; }
        private static object imageLocker = new object();
        void Temp(out  BitmapData bmData, out int bytesPerPixel, out int heightInPixels, out int widthInPixels)//zeby kazdy raz
nie pisac
        {
            bmData = imgBitMap.LockBits(new Rectangle(0, 0, imgBitMap.Width, imgBitMap.Height),
ImageLockMode.ReadOnly, imgBitMap.PixelFormat);
            bytesPerPixel = Bitmap.GetPixelFormatSize(imgBitMap.PixelFormat) / 8;
            heightInPixels = imgBitMap.Height;
            widthInPixels = imgBitMap.Width * bytesPerPixel;
        }
        public ImageManipulation(Bitmap imgBitmap)
        {
            this.imgBitMap = imgBitmap;
            this.imgBitmapCOPY = imgBitmap;

        }
        public byte GetPixel(int x, int y, BitmapData bmpData)
        {
            unsafe
            {
                byte* ptr = (byte*)((byte*)bmpData.Scan0 + (y * bmpData.Stride) + x);
                return *ptr;
            }
        }
        public void SetPixel(int x, int y, BitmapData bmpData, byte value)
        {
            unsafe
            {
                byte* ptr = (byte*)((byte*)bmpData.Scan0 + (y * bmpData.Stride) + x);
                *ptr = value;
            }
        }
        public void MaskOnImage(int[,] maskArray, uint K)
        {
            Convolution applyMask = new Convolution(maskArray, (int)K);
            applyMask.ApplyInPlace(imgBitMap);
        }
        public void MaskOnImage(int[,] maskArray)
        {
            Convolution applyMask = new Convolution(maskArray);
            applyMask.ApplyInPlace(imgBitMap);
        }
        public void StratchingRang(int Max, int Min)
        {
            BitmapData bmData = null;
            int bytesPerPixel, heightInPixels, widthInPixels;
            try
            {
                Temp(out  bmData, out  bytesPerPixel, out  heightInPixels, out  widthInPixels);
                for (int y = 0; y < heightInPixels; y++)
                {

                    for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                    {
                        if (Min <= GetPixel(x, y, bmData) && Max >= GetPixel(x, y, bmData))
                        {
                            SetPixel(x, y, bmData, (byte)((GetPixel(x, y, bmData) - Min) * (15 / (Max - Min))));
                        }
                        if (GetPixel(x, y, bmData) <= Min && GetPixel(x, y, bmData) > Max)
                        {
                            SetPixel(x, y, bmData, 0);
                        }
                    }
                }
                imgBitMap.UnlockBits(bmData);
            }
            catch
            {
                try
```

```
        {
            imgBitMap.UnlockBits(bmData);
            throw new Exception("Some problem in StratchingRange func");
        }
        catch
        {
            throw new Exception("Some problem in StratchingRange func");
        }
    }
}
public void Redukcja(int pi1, int pi2, int q1, int q2)
{
    BitmapData bmData = null;
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
        for (int y = 0; y < heightInPixels; y++)
        {
            for (int x = 0; x < widthInPixels; x += bytesPerPixel)
            {
                if (GetPixel(x, y, bmData) <= pi1)
                    SetPixel(x, y, bmData, (byte)q1);
                else if (GetPixel(x, y, bmData) > pi1 && GetPixel(x, y, bmData) <= pi2)
                {
                    SetPixel(x, y, bmData, (byte)q2);
                }
                else
                {
                    SetPixel(x, y, bmData, (byte)levels);
                }
            }
        }
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
            imgBitMap.UnlockBits(bmData);
            throw new Exception("Some problem in Redukcja func");
        }
        catch
        {
            throw new Exception("Some problem in Redukcja func");
        }
    }
}
public void TresholdMinMax(int Max, int Min)
{

    BitmapData bmData = null;
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
        for (int y = 0; y < heightInPixels; y++)
        {
            for (int x = 0; x < widthInPixels; x += bytesPerPixel)
            {
                if (GetPixel(x, y, bmData) < Min || GetPixel(x, y, bmData) > Max)
                {
                    SetPixel(x, y, bmData, 0);
                }
            }
        }
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
```

```
            imgBitMap.UnlockBits(bmData);
            throw new Exception("Some problem in TreshholdMinMax func");
        }
        catch
        {
            throw new Exception("Some problem in TreshholdMinMax func");
        }
    }

}
public void Treshold(int CurTresh)
{
    BitmapData bmData = null;
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
        for (int y = 0; y < heightInPixels; y++)
        {
            for (int x = 0; x < widthInPixels; x += bytesPerPixel)
            {
                if (GetPixel(x, y, bmData) < CurTresh)
                    SetPixel(x, y, bmData, 0);
                else
                    SetPixel(x, y, bmData, 255);
            }
        }

        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
            imgBitMap.UnlockBits(bmData);
            throw new Exception("Some problem in Treshhold func");
        }
        catch
        {
            throw new Exception("Some problem in Treshhold func");
        }
    }

}
public void GetHistogram(Histogram histogramy)
{
    BitmapData bmData = null;
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
        System.IntPtr Scan0 = bmData.Scan0;
        unsafe
        {
            byte* ptrFirstPixel = (byte*)(void*)Scan0;
            for (int y = 0; y < heightInPixels; y++)
            {
                byte* currentLine = ptrFirstPixel + (y * bmData.Stride);
                for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                {
                    switch (histogramy)
                    {
                        case Histogram.HistogramRGB:
                            histogramBlue[currentLine[x]]++;
                            histogramGreen[currentLine[x + 1]]++;
                            histogramRed[currentLine[x + 2]]++;
                            break;
                        case Histogram.Histogram:
                            histogramGray[currentLine[x]]++;
                            break;
                    }
                }
```

```
            }
        }
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
            imgBitMap.UnlockBits(bmData);
            throw new Exception("Some problem in Histogram func");
        }
        catch
        {
            throw new Exception("Some problem in Histogram func");
        }
    }
    ListOfHistogramGrey.Add(histogramGray);
    ListOfHistogramRGB.Add(histogramBlue);
    ListOfHistogramRGB.Add(histogramRed);
    ListOfHistogramRGB.Add(histogramGreen);
}
public MinMaxAveragePixel MaxMinPixelsMeth()
{
    BitmapData bmData = null;
    MinMaxAveragePixel MinMaxAverage = new MinMaxAveragePixel();
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
            for (int y = 0; y < heightInPixels; y++)
            {
                for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                {
                    MinMaxAverage.max = Math.Max(MinMaxAverage.max, GetPixel(x,y,bmData));
                    MinMaxAverage.min = Math.Min(MinMaxAverage.min, GetPixel(x,y,bmData));
                }
            }
        MinMaxAverage.averageG = histogramGreen.Average();
        MinMaxAverage.averageB = histogramBlue.Average();
        MinMaxAverage.averageR = histogramRed.Average();
        MinMaxAverage.averageGrayscale = histogramGray.Average();
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
            imgBitMap.UnlockBits(bmData);
            throw new Exception("Some problem in MinMaxAverage func");
        }
        catch
        {
            throw new Exception("Some problem in MinMaxAverage func");
        }
    }
    return MinMaxAverage;
}
public unsafe Bitmap Negate(Bitmap originalImage)
{
    Bitmap finalImage = new Bitmap(
    originalImage.Width,
    originalImage.Height,
    PixelFormat.Format24bppRgb);

    lock (imageLocker)
    {
        BitmapData originalImageData = originalImage.LockBits(
        new Rectangle(0, 0, finalImage.Width, finalImage.Height),
        ImageLockMode.ReadOnly,
        PixelFormat.Format24bppRgb);

        BitmapData finalImageData = finalImage.LockBits(
```

```
        new Rectangle(0, 0, finalImage.Width, finalImage.Height),
        ImageLockMode.WriteOnly,
        PixelFormat.Format24bppRgb);

        int originalImageRemain = originalImageData.Stride - originalImageData.Width * 3;
        int finalImageRemain = finalImageData.Stride - finalImageData.Width * 3;

        byte* originalImagePointer = (byte*)originalImageData.Scan0.ToPointer();
        byte* finalImagePointer = (byte*)finalImageData.Scan0.ToPointer();

        for (int i = 0; i < originalImage.Height; ++i)
        {
            for (int j = 0; j < originalImage.Width; ++j)
            {
                finalImagePointer[2] = (byte)(255 - originalImagePointer[2]);
                finalImagePointer[1] = (byte)(255 - originalImagePointer[1]);
                finalImagePointer[0] = (byte)(255 - originalImagePointer[0]);
                originalImagePointer += 3;
                finalImagePointer += 3;
            }
            originalImagePointer += originalImageRemain;
            finalImagePointer += finalImageRemain;
        }
        imgBitMap.UnlockBits(originalImageData);
        finalImage.UnlockBits(finalImageData);
    }

    return finalImage;
}
public unsafe Bitmap ConvertToGrayScale(Bitmap originalImage)
{
    Bitmap finalImage = new Bitmap(
    originalImage.Width,
    originalImage.Height,
    PixelFormat.Format8bppIndexed);

    //setting up a nice grayscale palette for a final image
    ColorPalette colorPalette = finalImage.Palette;
    for (int i = 0; i < 256; ++i)
        colorPalette.Entries[i] = Color.FromArgb(i, i, i);
    finalImage.Palette = colorPalette;

    lock (imageLocker)
    {
        BitmapData originalImageData = originalImage.LockBits(
        new Rectangle(0, 0, originalImage.Width, originalImage.Height),
        ImageLockMode.ReadOnly,
        PixelFormat.Format24bppRgb);

        BitmapData finalImageData = finalImage.LockBits(
        new Rectangle(0, 0, finalImage.Width, finalImage.Height),
        ImageLockMode.WriteOnly,
        PixelFormat.Format8bppIndexed);

        //getting number of bits which are used for pad the bitmap
        int originalImageRemain = originalImageData.Stride - originalImageData.Width * 3;
        int finalImageRemain = finalImageData.Stride - finalImageData.Width;

        //getting the pointers
        byte* originalImagePointer = (byte*)originalImageData.Scan0.ToPointer();
        byte* finalImagePointer = (byte*)finalImageData.Scan0.ToPointer();

        for (int i = 0; i < originalImage.Height; ++i)
        {
            for (int j = 0; j < originalImage.Width; ++j)
            {
                byte gray = (byte)(originalImagePointer[2] * 0.2989 +
                originalImagePointer[1] * 0.5870 +
                originalImagePointer[0] * 0.1140);

                finalImagePointer[0] = gray;
```

```
                originalImagePointer += 3;
                finalImagePointer += 1;
            }
            originalImagePointer += originalImageRemain;
            finalImagePointer += finalImageRemain;
        }
        originalImage.UnlockBits(originalImageData);
        finalImage.UnlockBits(finalImageData);
    }

    return finalImage;
}
public void equalizingHistogram(equalizationsMetods Metody)
{
    double[] NEW = new double[256];
    long[] Right = new long[256];
    long[] Left = new long[256];
    int R = 0;
    long H = 0;
    BitmapData bmData = null;
    histogramAVG = histogramGray.Average();
    //double[] histogramGray = new double[256];
    int bytesPerPixel, heightInPixels, widthInPixels;
    Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
    System.IntPtr Scan0 = bmData.Scan0;
    for (int z = 0; z <= 255; ++z)
    {
        Left[z] = Right[z];
        H += histogramGray[z];
        while (H > histogramAVG)
        {
            H -= (int)histogramAVG;
            R++;
        }
        Right[z] = R;
        switch (Metody)
        {
            case equalizationsMetods.metoda1:
                NEW[z] = (Right[z] + Left[z]) / 2;
                break;
            case equalizationsMetods.metoda2:
                NEW[z] = (Right[z] - Left[z]);
                break;
            case equalizationsMetods.metoda4:
                NEW[z] = (Right[z] + Left[z] / 2) + (Right[z] / 2);
                break;
            default:
                break;
        }
    }


    unsafe
    {
        Point[] neighbourPoints = new Point[] { new Point(1, 0), new Point(-1, 0), new Point(0, 1), new Point(0, -1), new
Point(1, 1), new Point(-1, -1), new Point(-1, 1), new Point(1, -1) };
        for (int i = 0; i < neighbourPoints.Length; i++)
        {
            neighbourPoints[i].X = neighbourPoints[i].X * bytesPerPixel;
        }
        byte* ptrFirstPixel = (byte*)(void*)Scan0;
        for (int y = 0; y < heightInPixels; y++)
        {
            byte* currentLine = ptrFirstPixel + (y * bmData.Stride);
            for (int x = 0; x < widthInPixels; x += bytesPerPixel)
            {
                if (Left[currentLine[x]] == Right[currentLine[x]])
                {
                    currentLine[x] = (byte)Left[currentLine[x]];
                }
                else
                {
```

```
                    Random rnd = new Random();
                    switch (Metody)
                    {
                       case equalizationsMetods.metoda1:
                          currentLine[x] = (byte)NEW[currentLine[x]];
                          break;
                       case equalizationsMetods.metoda2:
                          currentLine[x] = (byte)(rnd.Next(0, Convert.ToInt32(NEW[currentLine[x]])) + (Left[currentLine[x]]));
                          break;
                       case equalizationsMetods.metoda3:
                          int avg = 0, count = 0;
                          foreach (var Point in neighbourPoints)
                          {
                             if (x + Point.X >= 0 && x + Point.X < widthInPixels && y + Point.Y >= 0 && y + Point.Y <
heightInPixels)
                             {
                                byte* tempLine = ptrFirstPixel + ((y + Point.Y) * bmData.Stride);

                                avg += tempLine[x + Point.X];
                                ++count;
                             }
                          }
                          avg /= count;
                          if (avg > Right[currentLine[x]]) currentLine[x] = (byte)Right[currentLine[x]];
                          else if (avg < Left[currentLine[x]]) currentLine[x] = (byte)Left[currentLine[x]];
                          else currentLine[x] = (byte)avg;
                          break;
                       case equalizationsMetods.metoda4:
                          currentLine[x] = (byte)NEW[currentLine[x]];
                          break;
                       default:
                          break;
                    }


                 }
              }
           }
        }
        imgBitMap.UnlockBits(bmData);

     }
     public void OperationArythmeticandLogic(ArithmeticOperations operations)
     {
        BitmapData bmData1 = null;
        BitmapData bmData2 = null;
        BitmapData bmDataResult = null;
        Bitmap bit1 = null;
        Bitmap bit2 = null;

        try
        {
           bmData1 = imgBitMap.LockBits(new Rectangle(0, 0, imgBitMap.Width, imgBitMap.Height),
ImageLockMode.ReadOnly, imgBitMap.PixelFormat);
           bmData2 = imgBitmap2.LockBits(new Rectangle(0, 0, imgBitmap2.Width, imgBitmap2.Height),
ImageLockMode.ReadOnly, imgBitmap2.PixelFormat);
           int bytesPerPixel = Bitmap.GetPixelFormatSize(imgBitMap.PixelFormat) / 8;
           int heightInPixels = imgBitMap.Height > imgBitMap.Height ? imgBitMap.Height : imgBitMap.Height;
           int widthInPixels = imgBitmap2.Width > imgBitmap2.Width ? imgBitmap2.Width * bytesPerPixel :
imgBitMap.Width * bytesPerPixel;
           imgBitMap.UnlockBits(bmData1);
           imgBitmap2.UnlockBits(bmData2);
           bit1 = new Bitmap(imgBitMap, widthInPixels, heightInPixels);
           bit2 = new Bitmap(imgBitmap2, widthInPixels, heightInPixels);
           bmData1 = bit1.LockBits(new Rectangle(0, 0, bit1.Width, bit1.Height), ImageLockMode.ReadOnly,
bit1.PixelFormat);
           bmData2 = bit2.LockBits(new Rectangle(0, 0, bit2.Width, bit2.Height), ImageLockMode.ReadOnly,
bit2.PixelFormat);
           bmDataResult = imgBitmapCOPY.LockBits(new Rectangle(0, 0, imgBitmapCOPY.Width, imgBitmapCOPY.Height),
ImageLockMode.ReadOnly, imgBitmapCOPY.PixelFormat);
```

```
        System.IntPtr Scan01 = bmData1.Scan0;
        System.IntPtr Scan02 = bmData2.Scan0;
        System.IntPtr Scan0Res = bmDataResult.Scan0;
        unsafe
        {
            byte* ptrFirstPixel1 = (byte*)(void*)Scan01;
            byte* ptrFirstPixel2 = (byte*)(void*)Scan02;
            byte* ptrFirstPixelRes = (byte*)(void*)Scan0Res;
            for (int y = 0; y < heightInPixels; y++)
            {
                byte* currentLine1 = ptrFirstPixel1 + (y * bmData1.Stride);
                byte* currentLine2 = ptrFirstPixel2 + (y * bmData2.Stride);
                byte* currentLineRes = ptrFirstPixelRes + (y * bmDataResult.Stride);
                for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                {
                    switch (operations)
                    {
                        case ArithmeticOperations.OR:
                            currentLineRes[x] = (byte)(currentLine1[x] | currentLine2[x]);
                            break;
                        case ArithmeticOperations.AND:
                            currentLineRes[x] = (byte)(currentLine1[x] & currentLine2[x]);
                            break;
                        case ArithmeticOperations.ADD:
                            currentLineRes[x] = (byte)((currentLine1)[x] + currentLine2[x]);
                            break;
                        case ArithmeticOperations.XOR:
                            currentLineRes[x] = (byte)(currentLine1[x] ^ currentLine2[x]);
                            break;
                        case ArithmeticOperations.DIFF:
                            currentLineRes[x] = (byte)Math.Abs(currentLine1[x] - currentLine2[x]);
                            break;
                        case ArithmeticOperations.SUB:
                            currentLineRes[x] = (byte)(currentLine1[x] - currentLine2[x]);
                            break;
                        default:
                            break;
                    }

                }
            }
        }
        bit1.UnlockBits(bmData1);
        bit2.UnlockBits(bmData2);
        imgBitmapCOPY.UnlockBits(bmDataResult);
    }
    catch
    {
        try
        {
            bit1.UnlockBits(bmData1);
            bit2.UnlockBits(bmData2);
            imgBitmapCOPY.UnlockBits(bmDataResult);
        }
        catch
        {
        }
    }
}

public void StretchHistogram()
{
    BitmapData bmData = null;
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
        System.IntPtr Scan0 = bmData.Scan0;
        int multiplier = max != 0 ? 255 / (max - min) : 0;
        int newPixel;
            for (int y = 0; y < heightInPixels; y++)
            {
```

```
                for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                {
                    int oldPixel = GetPixel(x,y,bmData);
                    if (oldPixel <= min || oldPixel > max) newPixel = 0;
                    else newPixel = (oldPixel - min) * multiplier;
                    SetPixel(x,y,bmData, (byte)newPixel);
                }
            }

            imgBitMap.UnlockBits(bmData);
        }
        catch
        {
            try
            {
                imgBitMap.UnlockBits(bmData);
                throw new Exception("Error in stretch histogram function");

            }
            catch
            {
                throw new Exception("Error in stretch histogram function");

            }
        }
    }
    public void MedianOnImage(maskSize xSize, maskSize ySize, int xSizeInt, int ySizeInt, edgeMethods method, int
userPixel)
    {
        BitmapData bmData = null;
        Random rnd = new Random();
        Point[] points = new Point[xSizeInt * ySizeInt];
        int heightInPixels, widthInPixels, bytesPerPixel;
        try
        {
            Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
            //Lock it fixed with 32bpp
            System.IntPtr Scan0 = bmData.Scan0;
            unsafe
            {
                byte* ptrFirstPixel = (byte*)bmData.Scan0;
                for (int y = 0; y < heightInPixels; y++)
                {
                    byte* currentLine = ptrFirstPixel + (y * bmData.Stride);
                    for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                    {
                        if (!((y >= heightInPixels - 1 - (int)ySize) || (y <= (int)ySize) || (x >= widthInPixels - 1 - (int)xSize) || (x <=
(int)xSize)))
                        {

                            int[] neighbours = new int[xSizeInt * ySizeInt];
                            int a = 0;
                            for (int k = -xSizeInt / 2; k <= xSizeInt / 2; ++k)
                                for (int l = -ySizeInt / 2; l <= ySizeInt / 2; ++l)
                                {
                                    neighbours[a++] = Marshal.ReadByte((IntPtr)((byte*)Scan0 + ((y + l) * bmData.Stride) + (x +
k)));
                                }

                            Array.Sort(neighbours);
                            if (neighbours.Length % 2 == 1)
                                currentLine[x] = (byte)neighbours[neighbours.Length / 2];
                            else
                                currentLine[x] = (byte)Math.Min((neighbours[neighbours.Length / 2] +
neighbours[(neighbours.Length / 2) + 1]) / 2, levels - 1);
                        }
                        else
                        {
                            OnEdgeOperation(currentLine, x, method, userPixel);
                        }
                    }
                }
```

```
        }
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
            imgBitMap.UnlockBits(bmData);
        }
        catch
        {
        }
    }
}
public unsafe void OnEdgeOperationForMorph(edgeMethods methods, int userPixel)
{
    Random rnd = new Random();
    BitmapData bmData = null;
    int bytesPerPixel, heightInPixels, widthInPixels;
    try
    {
        //Lock it fixed with 32bpp
        Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
        System.IntPtr Scan0 = bmData.Scan0;
        unsafe
        {
            byte* ptrFirstPixel = (byte*)(void*)Scan0;
            for (int y = 0; y < heightInPixels; y++)
            {
                byte* currentLine = ptrFirstPixel + (y * bmData.Stride);
                for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                {
                    if (((y == heightInPixels - 1) || (y == 0) || (x == widthInPixels - 1 || (x == 0))))
                    {
                        OnEdgeOperation(currentLine, x, methods, userPixel);
                    }
                }
            }
        }
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
        try
        {
            imgBitMap.UnlockBits(bmData);
        }
        catch
        {
        }
    }
}
public unsafe void OnEdgeOperation(byte* currentLine, int x, edgeMethods method, int userPixel)
{
    Random rnd = new Random();
    unsafe
    {
        switch (method)
        {
            case edgeMethods.first:
                return;
                break;
            case edgeMethods.second:
                currentLine[x] = (byte)rnd.Next(0, 255);
                break;
            case edgeMethods.third:
                currentLine[x] = (byte)userPixel;
                break;
            case edgeMethods.mine:
                currentLine[x] = 0;
                break;
            default:
```

```
                    break;
                }
            }
        }
        public unsafe void MaskOnImage2(int userPixel, int[,] maskArray, edgeMethods methods, Graduation graduation)
        {
            Convolution applyMask = new Convolution(maskArray);
            applyMask.ApplyInPlace(imgBitMap);
            Random rnd = new Random();
            BitmapData bmData = null;
            int bytesPerPixel, heightInPixels, widthInPixels;
            try
            {
                Temp(out bmData, out bytesPerPixel, out heightInPixels, out widthInPixels);
                System.IntPtr Scan0 = bmData.Scan0;
                unsafe
                {
                    byte* ptrFirstPixel = (byte*)(void*)Scan0;
                    for (int y = 0; y < heightInPixels; y++)
                    {
                        byte* currentLine = ptrFirstPixel + (y * bmData.Stride);
                        for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                        {
                            if (((y == heightInPixels - 1) || (y == 0) || (x == widthInPixels - 1 || (x == 0))))
                            {
                                OnEdgeOperation(currentLine, x, methods, userPixel);
                            }
                        }
                    }
                    Graduation(graduation);
                }
                imgBitMap.UnlockBits(bmData);
            }
            catch
            {
                try
                {
                    imgBitMap.UnlockBits(bmData);
                }
                catch
                {
                }
            }
        }
        public void kompresjaREAD(Bitmap bitmap)
        {
            BitmapData bmData = null;

            const int PIXELLEN = 1;
            const int WORDLEN = 1;

            int newColor = 255, lastColor = 0;
            int repeatCount = 0;
            int total = 0;

            int width = bitmap.Width;
            int height = bitmap.Height;
            int fld = width * height;
            int before, after;
            float stopien;

            before = fld * PIXELLEN;
            try
            {
                //Lock it fixed with 32bpp
                bmData = imgBitMap.LockBits(new Rectangle(0, 0, imgBitMap.Width, imgBitMap.Height),
ImageLockMode.ReadOnly, imgBitMap.PixelFormat);

                //Przeglad wierszami
                for (int y = 0; y < height; y++)
                    for (int x = 0; x < width; x++)
                    {
```

```
        newColor = GetPixel(x, y, bmData);

        //Jesli ten sam kolor
        if (newColor == lastColor)
        {
           repeatCount++;
        }
        else
        {

           if (repeatCount > 0)
           {
              total += WORDLEN + PIXELLEN;
              repeatCount = 0;
           }

           total += WORDLEN + PIXELLEN;
        }

        lastColor = GetPixel(x, y, bmData);
     }

  if (repeatCount > 0)
  {
     total += WORDLEN + PIXELLEN;
     repeatCount = 0;
  }

  //Przeglad kolumnami
  for (int x = 0; x < width; x++)
     for (int y = 0; y < height; y++)
     {
        newColor = GetPixel(x, y, bmData);

        //Jesli ten sam kolor
        if (newColor == lastColor)
        {
           repeatCount++;
        }
        else
        {
           if (repeatCount > 0)
           {
              total += WORDLEN + PIXELLEN;
              repeatCount = 0;
           }
           total += WORDLEN + PIXELLEN;
        }
        lastColor = GetPixel(x, y, bmData);
     }

  if (repeatCount > 0)
  {
     total += WORDLEN + PIXELLEN;
     repeatCount = 0;
  }

  after = total / 2;
  stopien = (float)(float)before / (float)after;

  int bits = 0;
  for (int i = 1; i <= 8; i++)
  {
     int Levels = (int)Math.Pow(2, i);
     if (Levels < levels) continue;
     bits = i;
     break;
  }
  float div = bits / 8.0f;
  imgBitMap.UnlockBits(bmData);
  MessageBox.Show("\nRozmiar przed kompresją: " + before * div
                  + "\n\nRozmiar po kompresji: " + after * div
```

```
                    + "\n\nStopień kompresji: " + stopien, "Kompresja READ");
            }
            catch
            {
                try
                {
                    imgBitMap.UnlockBits(bmData);

                }
                catch
                {
                    throw new Exception("Can't Kompresion");
                }
            }

        }
        public Bitmap kompresjaBlokowa(Bitmap originalImage, int size)
        {
            int color = 0;
            float avg;
            float avgUP;
            float avgDOWN;
            int countUP;
            int countDOWN;
            int countAVG;
            int sizeAfter = 0;
            Bitmap bmp = (Bitmap)originalImage.Clone();
            BitmapData bmData = null;
            try
            {
                bmData = imgBitMap.LockBits(new Rectangle(0, 0, imgBitMap.Width, imgBitMap.Height),
ImageLockMode.ReadOnly, imgBitMap.PixelFormat);

                for (int y = 0; y < originalImage.Size.Height; y += size)
                    for (int x = 0; x < originalImage.Size.Width; x += size)
                    {
                        avg = 0;
                        countAVG = 0;
                        for (int yy = 0; yy < size; ++yy)
                        {
                            for (int xx = 0; xx < size; ++xx)
                            {
                                if (x + xx < originalImage.Size.Width && y + yy < originalImage.Size.Height)
                                {
                                    color = GetPixel(x + xx, y + yy, bmData);
                                    avg += color;
                                    ++countAVG;
                                }
                            }
                        }
                        avg /= countAVG;
                        avg = (int)avg;

                        avgUP = 0;
                        avgDOWN = 0;
                        countDOWN = 0;
                        countUP = 0;
                        for (int yy = 0; yy < size; ++yy)
                        {
                            for (int xx = 0; xx < size; ++xx)
                            {
                                if (x + xx < originalImage.Size.Width && y + yy < originalImage.Size.Height)
                                {
                                    color = GetPixel(x + xx, y + yy, bmData);
                                    if (color >= avg) { avgUP += color; ++countUP; }
                                    else { avgDOWN += color; ++countDOWN; }
                                }
                            }
                        }
                        avgUP /= countUP;
                        avgDOWN /= countDOWN;
                        avgUP = (int)avgUP;
```

```
                avgDOWN = (int)avgDOWN;

                for (int yy = 0; yy < size; ++yy)
                    for (int xx = 0; xx < size; ++xx)
                        if (x + xx < originalImage.Size.Width && y + yy < originalImage.Size.Height)
                            if (GetPixel(x + xx, y + yy, bmData) >= avg) SetPixel(x + xx, y + yy, bmData, (byte)avgUP);
                            else SetPixel(x + xx, y + yy, bmData, (byte)avgDOWN);

                sizeAfter += countAVG + 16;
            }

            int sizeBefore = originalImage.Size.Width * originalImage.Size.Height * 8;

            int bits = 0;
            for (int i = 1; i <= 8; i++)
            {
                int Levels = (int)Math.Pow(2, i);
                if (Levels < levels) continue;
                bits = i;
                break;
            }
            float div = bits / 8.0f;
            imgBitMap.UnlockBits(bmData);
            MessageBox.Show("\nRozmiar przed kompresją: " + sizeBefore * div
                    + "\n\nRozmiar po kompresji: " + sizeAfter * div
                    + "\n\nStopień kompresji: " + (float)sizeBefore / (float)sizeAfter, "Kompresja blokowa");

        }
        catch
        {
            try
            {
                imgBitMap.UnlockBits(bmData);
            }
            catch
            {
                imgBitMap.UnlockBits(bmData);
            }
        }


        return bmp;
    }

    public void kompresjaHuffman(Bitmap originalImage)
    {

        int x = 2;
        int licznik = 0;
        int przed;
        int po = 0;
        float stopien;
        int moc = 2;
        int ilosc = 0;
        BitmapData bmData = null;
        int[] phist = new int[levels];
        przed = originalImage.Width * originalImage.Height;
        try
        {
            bmData = imgBitMap.LockBits(new Rectangle(0, 0, imgBitMap.Width, imgBitMap.Height),
ImageLockMode.ReadOnly, imgBitMap.PixelFormat);

            for (int y = 0; y < originalImage.Height; y++)
                for (int xx = 0; xx < originalImage.Width; xx++)
                    phist[GetPixel(xx, y, bmData)] += 1;

            for (int i = 0; i < levels; i++)
            {
                if (phist[i] != 0)
                {
                    licznik++;
                    po += (licznik * phist[i]);
```

```
            if (licznik >= Math.Pow((double)x, (double)moc))
            {
               licznik = 0;
               moc++;
            }
            ilosc++;
         }

         if (licznik == 1)
         {
            po = przed + 1;
         }
      }

      po /= 8;
      po += (ilosc * 2);

      stopien = (float)(float)przed / (float)po;

      int bits = 0;
      for (int i = 1; i <= 8; i++)
      {
         int Levels = (int)Math.Pow(2, i);
         if (Levels < levels) continue;
         bits = i;
         break;
      }
      float div = bits / 8.0f;
      imgBitMap.UnlockBits(bmData);
      MessageBox.Show("\nRozmiar przed kompresją: " + przed * div
      + "\n\nRozmiar po kompresji: " + po * div
      + "\n\nStopień kompresji: " + stopien, "Kompresja Huffman");

   }
   catch
   {
      try
      {
         imgBitMap.UnlockBits(bmData);
      }
      catch
      {

      }
   }
}
public unsafe void Graduation(Graduation graduation)
      {
         Random rnd = new Random();
         BitmapData bmData = null;
         int bytesPerPixel, heightInPixels, widthInPixels;
         try
         {
      //Lock it fixed with 32bpp
            Temp(out bmData,out bytesPerPixel,out  heightInPixels,out  widthInPixels);
            System.IntPtr Scan0 = bmData.Scan0;
            unsafe
            {
               byte* ptrFirstPixel = (byte*)(void*)Scan0;
               for (int y = 0; y < heightInPixels; y++)
               {
                  byte* currentLine = ptrFirstPixel + (y * bmData.Stride);
                  for (int x = 0; x < widthInPixels; x += bytesPerPixel)
                  {
                     switch (graduation)
                     {
                        case Csharp.Graduation.first:
                           currentLine[x] = (byte)((currentLine[x] - min) / (max - min) * (levels - 1));
                           break;
                        case Csharp.Graduation.second:
                           if (currentLine[x] < 0)
```

```
                        {
                            currentLine[x] = 0;
                        }
                        else if (currentLine[x] > 0)
                        {
                            currentLine[x] = (byte)(levels - 1);
                        }
                        else
                        {
                            currentLine[x] = (byte)(levels - 1);
                        }

                        break;
                    case Csharp.Graduation.third:
                        if (currentLine[x] < 0)
                        {
                            currentLine[x] = 0;
                        }
                        else if (0 <= currentLine[x] && currentLine[x] <= levels - 1)
                        {
                            currentLine[x] = currentLine[x];
                        }
                        else if (currentLine[x] > levels - 1)
                        {
                            currentLine[x] = (byte)(levels - 1);
                        }
                        break;
                }
            }
        }
        imgBitMap.UnlockBits(bmData);
    }
}
catch
{
    try
    {
        imgBitMap.UnlockBits(bmData);
    }
    catch
    {
    }
}
}

    }
}
```

FORM1.cs

```
public partial class Form1 : Form
  {
    int userPixel = 0;
    ImageManipulation imageClass;
    public Form1()
    {
        InitializeComponent();
        histogramToolStripMenuItem.Enabled = false;
        grayscalePicture2ToolStripMenuItem.Enabled = false;
        grayScaleToolStripMenuItem2.Enabled = false;
        masksToolStripMenuItem.Enabled = false;
        secondToolStripMenuItem.Enabled = false;
        processingToolStripMenuItem.Enabled = false;
        grayScaleToolStripMenuItem2.Enabled = false;
        grayscalePicture2ToolStripMenuItem.Enabled = false;
        operationsWithTwoPicturesToolStripMenuItem1.Visible = false;
        operationsWithTwoPicturesToolStripMenuItem.Visible = false;
    }
    private string OpenImage()
    {
        openFileDialog1.Filter = "";

        ImageCodecInfo[] codecs = ImageCodecInfo.GetImageEncoders();
        string sep = string.Empty;
```

```
            foreach (var c in codecs)
            {
                string codecName = c.CodecName.Substring(8).Replace("Codec", "Files").Trim();
                openFileDialog1.Filter = String.Format("{0}{1}{2} ({3})|{3}", openFileDialog1.Filter, sep, codecName,
c.FilenameExtension);
                sep = "|";
            }
            openFileDialog1.Filter = String.Format("{0}{1}{2} ({3})|{3}", openFileDialog1.Filter, sep, "All Files", "*.*");
            openFileDialog1.DefaultExt = ".png";

            DialogResult result = openFileDialog1.ShowDialog();

            if (result == DialogResult.OK)
            {
                return openFileDialog1.FileName;
            }
            return "fail";
        }
        private void firstToolStripMenuItem_Click(object sender, EventArgs e)
        {
            try
            {
                string fileName = OpenImage();
                if (fileName == "fail")
                    return;
                Bitmap image1 = new Bitmap(fileName, true);
                imageClass = new ImageManipulation(image1);
                new System.Threading.Thread(PictireForm1).Start();
                secondToolStripMenuItem.Enabled = true;
                grayScaleToolStripMenuItem1.Enabled = false;
                masksToolStripMenuItem.Enabled = false;
                histogramToolStripMenuItem.Enabled = false;
                processingToolStripMenuItem.Enabled = true;
                grayScaleToolStripMenuItem1.Enabled = false;
                grayScaleToolStripMenuItem2.Enabled = true;
                histogramToolStripMenuItem.Enabled = true;
                rGBToolStripMenuItem.Enabled = true;
            }
            catch (Exception exception)
            {

                MessageBox.Show(exception.Message, "Exception");
            }
        }
        private void secondToolStripMenuItem_Click(object sender, EventArgs e)
        {
            try
            {
                string fileName = OpenImage();
                if (fileName == "fail")
                    return;
                imageClass.imgBitmap2 = new Bitmap(fileName);
                new System.Threading.Thread(PictireForm2).Start();
                grayscalePicture2ToolStripMenuItem.Enabled = true;
                masksToolStripMenuItem.Enabled = false;
            }
            catch (Exception exception)
            {
                MessageBox.Show(exception.Message, "Exception");
            }
        }
        void ResultPicture()
        {
            ResultPicture RP = new ResultPicture(imageClass.imgBitMap);
            RP.ShowDialog();
        }
        void PictireForm1()
        {
            PC1 pc1 = new PC1(imageClass.imgBitMap);
            pc1.ShowDialog();
        }
        void PictireForm2()
```

```
{
    PC2 pc2 = new PC2(imageClass.imgBitmap2);
    pc2.ShowDialog();
}
void MasksForm()
{
    Operations masks = new Operations(ref imageClass);
    masks.mask_imageClass = imageClass;
    masks.ShowDialog();
}
void GrayscaleHistogram()
{
    GrayscaleHistogram grayscaleHistogram = new GrayscaleHistogram(ref imageClass);
    grayscaleHistogram.GrayscaleHistogram_imageClass = imageClass;
    grayscaleHistogram.ShowDialog();
}
private void grayScaleToolStripMenuItem1_Click(object sender, EventArgs e)
{
    new System.Threading.Thread(GrayscaleHistogram).Start();
}
private void restartToolStripMenuItem_Click(object sender, EventArgs e)
{
    Application.Restart();
}
private void grayScaleToolStripMenuItem2_Click(object sender, EventArgs e)
{
    imageClass.imgBitMap = imageClass.ConvertToGrayScale(imageClass.imgBitMap);
    imageClass.imgBitmapCOPY = imageClass.ConvertToGrayScale(imageClass.imgBitmapCOPY);
    rGBToolStripMenuItem.Enabled = false;
    grayScaleToolStripMenuItem1.Enabled = true;
    masksToolStripMenuItem.Enabled = true;
    new System.Threading.Thread(PictireForm1).Start();
}
private void grayscalePicture2ToolStripMenuItem_Click(object sender, EventArgs e)
{
    imageClass.imgBitmap2 = imageClass.ConvertToGrayScale(imageClass.imgBitmap2);
    operationsWithTwoPicturesToolStripMenuItem.Enabled = true;
    new System.Threading.Thread(PictireForm2).Start();
}
private void infoToolStripMenuItem_Click(object sender, EventArgs e)
{
    MessageBox.Show(" Autor: Taras Kuts");
}
    void rgbhistogram()
{
    RGBhistogram rgbhistogram = new RGBhistogram(ref imageClass);
    rgbhistogram.RGBhistogram_imageClass = imageClass;
    rgbhistogram.ShowDialog();
}
private void rGBToolStripMenuItem_Click(object sender, EventArgs e)
{
    new System.Threading.Thread(rgbhistogram).Start();

}
private void masksToolStripMenuItem_Click(object sender, EventArgs e)
{
    MasksForm();
}
private void sUBToolStripMenuItem1_Click(object sender, EventArgs e)
{
    Subtract filtersub = new Subtract(imageClass.imgBitMap);
    imageClass.imgBitmapCOPY = filtersub.Apply(imageClass.imgBitmap2);
    ResultPicture result = new ResultPicture(imageClass.imgBitmapCOPY);
    result.ShowDialog();
}

private void xORToolStripMenuItem1_Click(object sender, EventArgs e)
{
    imageClass.OperationArythmeticandLogic(ArithmeticOperations.XOR);
    ResultPicture resultP = new ResultPicture(imageClass.imgBitmapCOPY);
    resultP.ShowDialog();
}
```

```
        private void aNDToolStripMenuItem1_Click(object sender, EventArgs e)
        {
          imageClass.OperationArythmeticandLogic(ArithmeticOperations.AND);
            ResultPicture resultP = new ResultPicture(imageClass.imgBitmapCOPY);
            resultP.ShowDialog();
        }

        private void oRToolStripMenuItem1_Click(object sender, EventArgs e)
        {
             imageClass.OperationArythmeticandLogic(ArithmeticOperations.OR);
            ResultPicture resultP = new ResultPicture(imageClass.imgBitmapCOPY);
            resultP.ShowDialog();
        }

        private void aDDanotherToolStripMenuItem_Click(object sender, EventArgs e)
        {
             imageClass.OperationArythmeticandLogic(ArithmeticOperations.ADD);
            ResultPicture resultP = new ResultPicture(imageClass.imgBitmapCOPY);
            resultP.ShowDialog();
        }

        private void sUBanotherToolStripMenuItem_Click(object sender, EventArgs e)
        {
            imageClass.OperationArythmeticandLogic(ArithmeticOperations.SUB);
            ResultPicture resultP = new ResultPicture(imageClass.imgBitmapCOPY);
            resultP.ShowDialog();
        }

        private void dIFFERENCEToolStripMenuItem2_Click(object sender, EventArgs e)
        {
            imageClass.OperationArythmeticandLogic(ArithmeticOperations.DIFF);
            ResultPicture resultP = new ResultPicture(imageClass.imgBitmapCOPY);
            resultP.ShowDialog();
        }
    }
}
```

operation.cs

```
  partial class Operations : Form
    {
        public ImageManipulation mask_imageClass { get; set; }
        int Max,Min;
        uint kDiv = 0;
        int userPixel = 0;
        int ComprSize;
        int TreshCur;
        int Pi1, Pi2, Q1, Q2;
        int medUserPixel = 0;
        short[,] rombs = { { 0, 1, 0 }, { 1, 1, 1 }, { 0, 1, 0 } };
        short[,] square = { { 1, 1, 1 }, { 1, 1, 1 }, { 1, 1, 1 } };
        List<TextBox> txtMaskMatrixList = new List<TextBox>();
        List<TextBox> txtEdgeMatrixList = new List<TextBox>();
        List<TextBox> txtGraduationMatrixList = new List<TextBox>();
        List<TextBox> txtSize1 = new List<TextBox>();
        List<TextBox> txtSize2 = new List<TextBox>();
        List<TextBox> txtEdgeList = new List<TextBox>();
        List<int[,]> intMaskList = new List<int[,]>();
        List<int[,]> intEdgeList = new List<int[,]>();
        List<int[,]> intGraduationList = new List<int[,]>();
        int maskSizeX = 3, maskSizeY = 3;
        masksAplied mask;
        public Operations(ref ImageManipulation ImageClass)
        {
            mask_imageClass = ImageClass;
            InitializeComponent();
            rdReductionwithpar.Enabled = false;
            rdStratchingMinMax.Enabled = false;
            rdTreshholdMinMax.Enabled = false;
            rdTreshhold.Enabled = false;
            pictureBox1.Image = ImageClass.imgBitMap;
```

```
            init();
            initMasks();
        }
        void parseUserPixel()
        {
            int userPixel;
            Int32.TryParse(txtUserPixel.Text, out userPixel);
        }
        private void initMasks()
        {
            intMaskList.Add(new int[3, 3] { { 1, 2, 1 }, { 2, 4, 2 }, { 1, 2, 1 } }); //onNine
            intMaskList.Add(new int[3, 3] { { 1, 1, 1 }, { 1, 2, 1 }, { 1, 1, 1 } });//onTen
            intMaskList.Add(new int[3, 3] { { 1, 2, 1 }, { 2, 4, 2 }, { 1, 2, 1 } });//onSixteen
            intMaskList.Add(new int[3, 3] { { 0, 1, 0 }, { 1, -4, 1 }, { 0, 1, 0 } });//cyfrowaLaplas
            intMaskList.Add(new int[3, 3] { { 0, -1, 0 }, { -1, 4, -1 }, { 0, -1, 0 } });//laplasA
            intMaskList.Add(new int[3, 3] { { -1, -1, -1 }, { -1, 8, -1 }, { -1, -1, -1 } });//laplasB
            intMaskList.Add(new int[3, 3] { { 1, -2, 1 }, { -2, 4, -2 }, { 1, -2, 1 } });//laplasC
            intMaskList.Add(new int[3, 3] { { -1, -1, -1 }, { -1, 9, -1 }, { -1, -1, -1 } });//laplasD
            intMaskList.Add(new int[3, 3] { { 1, -2, 1 }, { -2, 5, -2 }, { 1, -2, 1 } });//edgeOne
            intMaskList.Add(new int[3, 3] { { -1, -1, -1 }, { -1, 9, -1 }, { -1, -1, -1 } });//edgeTwo
            intMaskList.Add(new int[3, 3] { { 0, -1, 0 }, { -1, 5, -1 }, { 0, -1, 0 } });//edgeThree
            intMaskList.Add(new int[3, 3] { { -1, -2, -1 }, { 0, 0, 0 }, { 1, 2, 1 } });//Sobela
            intMaskList.Add(new int[3, 3] { { -1, 0, 1 }, { -2, 0, 2 }, { -1, 0, 1 } });//Sobela2
            intMaskList.Add(new int[3, 3] { { -1, 0, 1 }, { 1, 0, 1 }, { 1, 0, 1 } });//Prewitt
            intMaskList.Add(new int[3, 3] { { -1, -1, -1 }, { 0, 0, 0 }, { 1, 1, 1 } });//Prewitt2
            intMaskList.Add(new int[3, 3] { { -2, -2, -2 }, { 1, 0, 1 }, { 3, 1, 3 } });//Universal
            intMaskList.Add(new int[3, 3] { { -4, -4, -1 }, { 2, 2, 2 }, { 4, 4, 4 } });//Universal2
        }
        void init()
        {
            string[] masks = new string[] { "Wygladzanie 1/9", "Wygladzanie 1/10", "Wygladzanie 1/16", "Wyostrzanie
cyfroweLaplasa", "LaplasA", "LaplasB", "LaplasC", "LaplasD", "edgeOne", "edgeTwo", "edgeThree", "Sobela", "Sobela2",
"Prewitt", "Prewitt2", "Universal", "Universal2" };
            foreach (var item in masks) { cbMasks.Items.Add(item); }
            string[] edges = new string[] { "Nothing to change","Random","UserPixel","Black" };
            foreach (var item in edges) { cbEdge.Items.Add(item); }
            string[] graduations = new string[] { "Grad1", "Grad2", "Grad3" };
            foreach (var item in graduations) { cbGraduation.Items.Add(item); }
            string[] edges2 = new string[] { "NothingToChange", "Random", "Your","Black" };
            foreach (var item in edges2) { cmEdge2.Items.Add(item); }
            string[] Size1 = new string[] { "3", "5", "7"};
            foreach (var item in Size1) { cmFirstSize.Items.Add(item); }
            string[] Size2 = new string[] { "3", "5", "7"};
            foreach (var item in Size2) { cmSecondSize.Items.Add(item); }

            // }
            var query = tabCtrlLab.TabPages[0].Controls.Cast<Control>().OrderBy((ctrl) => ctrl.Location.Y).ThenBy((ctrl) =>
ctrl.Location.X).Where((ctrl) => ctrl is TextBox && ctrl.Name.Substring(0, 7) == "txtMask").Select((ctrl) => ctrl as TextBox);
            foreach (var item in query)
            {
                txtMaskMatrixList.Add(item);
            }

        }
        private void cbMasks_SelectedIndexChanged_1(object sender, EventArgs e)
        {
            mask = (masksAplied)cbMasks.SelectedIndex;
            int k = 0;
            for (int i = 0; i < maskSizeX; i++)
            {
                for (int j = 0; j < maskSizeY; j++)
                {
                    txtMaskMatrixList[k].Text = Convert.ToString(intMaskList[(int)mask].GetValue(i, j));
                    k++;
                }
            }
        }
        private void btnApplyMask_Click_1(object sender, EventArgs e)
        {
            if (cbOwnMask.Checked)
            {
                try
```

```
            {
                int k = 0;
                int[,] ownMask = new int[maskSizeX, maskSizeY];
                for (int i = 0; i < maskSizeX; i++)
                {
                    for (int j = 0; j < maskSizeY; j++)
                    {
                        ownMask[i, j] = Convert.ToInt32(txtMaskMatrixList[k].Text);
                        k++;
                    }
                }
                mask_imageClass.MaskOnImage(ownMask);
            }
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message, "Exception");
            }
        }
        else if (mask <= masksAplied.onSixteen)
        {
            kDiv = 16;
            mask_imageClass.MaskOnImage(intMaskList[(int)mask], kDiv);
        }
        else if (mask <= masksAplied.onNine)
        {
            kDiv = 9;
            mask_imageClass.MaskOnImage(intMaskList[(int)mask], kDiv);
        }
        else if (mask <= masksAplied.onTen)
        {
            kDiv = 10;
            mask_imageClass.MaskOnImage(intMaskList[(int)mask], kDiv);
        }
        else { mask_imageClass.MaskOnImage2((int)userPixel, intMaskList[(int)mask],
(edgeMethods)cbEdge.SelectedIndex, (Graduation)cbGraduation.SelectedIndex); }
        pictureBox1.Image = mask_imageClass.imgBitMap;
    }
    private void pictureBox1_Click(object sender, EventArgs e)
    {
        ResultPicture resultPicture = new ResultPicture(mask_imageClass.imgBitMap);
        resultPicture.ShowDialog();
    }
    private void cbOwnMask_CheckedChanged(object sender, EventArgs e)
    {
        foreach (var item in txtMaskMatrixList)
        {
            item.ReadOnly = !cbOwnMask.Checked;
        }
    }
    private void Txbmin_TextChanged(object sender, EventArgs e)
    {
        rdTreshholdMinMax.Enabled = (Int32.TryParse(Txbmax.Text, out Max) && (Int32.TryParse(Txbmin.Text, out Min)));
        rdStratchingMinMax.Enabled = (Int32.TryParse(Txbmax.Text, out Max) && (Int32.TryParse(Txbmin.Text, out Min)));
    }
        private void txtTreshhold_TextChanged(object sender, EventArgs e)
    {
        rdTreshhold.Enabled = (Int32.TryParse(txtTreshhold.Text, out TreshCur));
    }
    private void txtPi2_TextChanged(object sender, EventArgs e)
    {
        rdReductionwithpar.Enabled = (Int32.TryParse(txtPi1.Text, out Pi1) && Int32.TryParse(txtPi2.Text, out Pi2) &&
Int32.TryParse(txtQ1.Text, out Q1) && Int32.TryParse(txtQ2.Text, out Q2));
    }
    private void txtQ1_TextChanged(object sender, EventArgs e)
    {
        rdReductionwithpar.Enabled = (Int32.TryParse(txtPi1.Text, out Pi1) && Int32.TryParse(txtPi2.Text, out Pi2) &&
Int32.TryParse(txtQ1.Text, out Q1) && Int32.TryParse(txtQ2.Text, out Q2));
    }

    private void txtComressionPar_TextChanged(object sender, EventArgs e)
    {
        rdCompressionPar.Enabled = (Int32.TryParse(txtComressionPar.Text, out ComprSize));
```

```
        }

        private void btnCompression_Click(object sender, EventArgs e)
        {
            if (rdCodeHuff.Checked == true) { mask_imageClass.kompresjaHuffman(mask_imageClass.imgBitMap); }
            else if (rdRead.Checked == true) { mask_imageClass.kompresjaREAD(mask_imageClass.imgBitMap); }
            else if(rdReductionwithpar.Checked == true)
            {
                if(ComprSize < 0 || ComprSize > 32) { MessageBox.Show("Enter number in range 0-32"); }
                else
                {
                    mask_imageClass.kompresjaBlokowa(mask_imageClass.imgBitMap, ComprSize);
                }
            }
        }
        void CountingMeanOfGray()
        {
            mask_imageClass.GetHistogram(Histogram.Histogram);
        }
        void ShowPicture()
        {
            pictureBox1.Image = mask_imageClass.imgBitMap;
        }

        private void btnApplyEqua_Click(object sender, EventArgs e)
        {
            CountingMeanOfGray();
            if (rdEqua1.Checked == true) { mask_imageClass.equalizingHistogram(equalizationsMetods.metoda1);
pictureBox1.Image = mask_imageClass.imgBitMap; ShowPicture(); }
            else if (rdEqua2.Checked == true) { mask_imageClass.equalizingHistogram(equalizationsMetods.metoda2);
ShowPicture(); }
            else if (rdEqua3.Checked == true) { mask_imageClass.equalizingHistogram(equalizationsMetods.metoda3);
ShowPicture(); }
            else if (rdEqua4.Checked == true) { mask_imageClass.equalizingHistogram(equalizationsMetods.metoda4);
ShowPicture(); }
            else if (rdStretching.Checked == true) { mask_imageClass.equalizingHistogram(equalizationsMetods.metoda4);
ShowPicture(); }
        }

        private void btnApplyMediana_Click(object sender, EventArgs e)
        {
            mask_imageClass.MedianOnImage((maskSize)cmFirstSize.SelectedIndex, (maskSize)cmFirstSize.SelectedIndex, 3,
3, (edgeMethods)cmEdge2.SelectedIndex, medUserPixel);
            ShowPicture();
        }

        private void txtUserPixel1_TextChanged(object sender, EventArgs e)
        {
            if(cmEdge2.SelectedIndex == 2)
            {
                Int32.TryParse(txtUserPixel1.Text, out medUserPixel);
            }
        }

        private void txtQ2_TextChanged(object sender, EventArgs e)
        {
            rdReductionwithpar.Enabled = (Int32.TryParse(txtPi1.Text, out Pi1) && Int32.TryParse(txtPi2.Text, out Pi2) &&
Int32.TryParse(txtQ1.Text, out Q1) && Int32.TryParse(txtQ2.Text, out Q2));
        }
        private void btnApllyFilter_Click(object sender, EventArgs e)
        {
            if (rdDilation.Checked == true)
            {
                Dilatation dilation = new Dilatation();
                dilation.ApplyInPlace(mask_imageClass.imgBitMap);
                pictureBox1.Image = mask_imageClass.imgBitMap;
            }
            else if (rdErode.Checked == true)
            {
                Erosion erosion = new Erosion();
                erosion.ApplyInPlace(mask_imageClass.imgBitMap);
                pictureBox1.Image = mask_imageClass.imgBitMap;
```

```
        }
        else if (rdOpen.Checked == true)
        {
            Opening opening = new Opening();
            opening.ApplyInPlace(mask_imageClass.imgBitMap);
            pictureBox1.Image = mask_imageClass.imgBitMap;
        }
        else if (rdClosed.Checked == true)
        {
            Closing closing = new Closing();
            closing.ApplyInPlace(mask_imageClass.imgBitMap);
            pictureBox1.Image = mask_imageClass.imgBitMap;
        }
        else if(rdNegative.Checked == true)
        {
            mask_imageClass.imgBitMap = mask_imageClass.Negate(mask_imageClass.imgBitMap);
            pictureBox1.Image = mask_imageClass.imgBitMap;
        }
        else if(rdReductionwithpar.Checked == true)
        {
            if (Q1 < 0 || Q1 > 255 || Q2 < 0 || Q2 > 255 || Pi1 < 0 || Pi1 > 255 || Pi2 < 0 || Pi2 > 255)
{ MessageBox.Show("Enter numbers in range 0 - 255"); }
            else
            {
                mask_imageClass.Redukcja(Pi1, Pi2, Q1, Q2);
                pictureBox1.Image = mask_imageClass.imgBitMap;
            }
        }
        else if(rdStratchingMinMax.Checked == true)
        {
            if (Min < 0 || Min > 255)
            {
                MessageBox.Show("Min gets numbers from  1 to 255");
            }
            else if (Min <= 0 || Min > 254)
            {
                MessageBox.Show("Max gets numbers form 0 to 254");
            }
            else
            {
                mask_imageClass.StratchingRang(Max, Min);
                pictureBox1.Image = mask_imageClass.imgBitMap;
            }
        }
        else if(rdTreshholdMinMax.Checked == true)
        {
            if (Min < 0 || Min > 255)
            {
                MessageBox.Show("Min gets numbers from  1 to 255");
            }
            else if (Min <= 0 || Min > 254)
            {
                MessageBox.Show("Max gets numbers form 0 to 254");
            }
            else
            {
                mask_imageClass.TresholdMinMax(Max, Min);
                pictureBox1.Image = mask_imageClass.imgBitMap;
            }
        }
        else if(rdTreshhold.Checked == true)
        {
            if (TreshCur < 0 || TreshCur > 255) MessageBox.Show("The threshold value is in the [0, 255] range");
            else { mask_imageClass.Treshold(TreshCur); }
            pictureBox1.Image = mask_imageClass.imgBitMap;
        }
        else
        {
            MessageBox.Show("Choose one of the filters first");
        }
    }
    private void txtPi1_TextChanged(object sender, EventArgs e)
```

```
        {
            rdReductionwithpar.Enabled = (Int32.TryParse(txtPi1.Text, out Pi1) && Int32.TryParse(txtPi2.Text,out Pi2) &&
Int32.TryParse(txtQ1.Text, out Q1) && Int32.TryParse(txtQ2.Text,out  Q2));
        }

        private void Txbmax_TextChanged(object sender, EventArgs e)
        {
            rdStratchingMinMax.Enabled = (Int32.TryParse(Txbmax.Text, out Max) && (Int32.TryParse(Txbmin.Text,out Min)));
            rdStratchingMinMax.Enabled = (Int32.TryParse(Txbmax.Text, out Max) && (Int32.TryParse(Txbmin.Text, out Min)));
        }

        private void txtMask1_1_TextChanged(object sender, EventArgs e)
        {
            int temp;
            btnApplyMask.Enabled = Int32.TryParse((sender as TextBox).Text, out temp);
        }


    }
}
```

GrayscaleHistogram.cs

```
namespace Csharp
{
    partial class GrayscaleHistogram : Form
    {
        public ImageManipulation GrayscaleHistogram_imageClass { get; set; }
        MinMaxAveragePixel MinMaxAveragePixel = new MinMaxAveragePixel();
        public GrayscaleHistogram(ref ImageManipulation ImageClass)
        {
            GrayscaleHistogram_imageClass = ImageClass;
            InitializeComponent();
            HistogramShow();

        }
        public void HistogramShow()
        {
            GrayscaleHistogram_imageClass.GetHistogram(Histogram.Histogram);
            MinMaxAveragePixel = GrayscaleHistogram_imageClass.MaxMinPixelsMeth();
            histogramGrayscale.Invalidate();
            txtMax.Text = MinMaxAveragePixel.max.ToString();
            txtMin.Text = MinMaxAveragePixel.min.ToString();
            txtMean.Text = MinMaxAveragePixel.averageGrayscale.ToString();
            List<long> histogramGray = GrayscaleHistogram_imageClass.ListOfHistogramGrey[0].ToList<long>();
            histogramGrayscale.Series["Grey"].Points.DataBindXY(Enumerable.Range(1, 256).ToList(), histogramGray);
            Axis ax = histogramGrayscale.ChartAreas[0].AxisX;
            ax.Minimum = 0;
            ax.Maximum = 255;
        }


    }
}
```

MinMaxAveragePixel.cs

```
    class MinMaxAveragePixel
    {
        public int min, max;
        public double averageGrayscale;
        public double averageR, averageB, averageG;
        public MinMaxAveragePixel()
        {
            averageB = 0;
            averageG = 0;
            averageR = 0;
            averageGrayscale = 0;
            min = 255;
            max = 0;
        }
    }
```

RGBhistogram.cs prawie taki sam

PC1.cs

```
public partial class PC1 : Form
   {
      int prevHeight;
      int prevWidth;
      private Bitmap _previewBitmap;
      public PC1(Bitmap bitmap)
      {
         if (bitmap == null) {return;}
         InitializeComponent();
         _previewBitmap = bitmap;
         pictureBox1.Image = _previewBitmap;
      }
     private void zoomINToolStripMenuItem_Click(object sender, EventArgs e)
      {
         prevHeight = _previewBitmap.Height;
         prevWidth = _previewBitmap.Width;
         int zoom = 10;
         if (zoom < 20)
         {
            this.pictureBox1.Width += (int)(prevWidth * 0.1);
            this.pictureBox1.Height += (int)(prevHeight * 0.1);
            zoom++;
            this.AutoSize = true;
         }
      }

      private void zoomOUTToolStripMenuItem_Click(object sender, EventArgs e)
      {
         int zoom = 10;
         if (zoom > 0)
         {
            this.pictureBox1.Width -= (int)(this.pictureBox1.Width * 0.1);
            this.pictureBox1.Height -= (int)(this.pictureBox1.Height * 0.1);
            zoom--;
            this.Size = new Size(pictureBox1.Width, pictureBox1.Height);
         }
      }
   }
```

PC2.cs analogiczny