LAMPIRAN A

A. AForge.Net Framework

A.1 Grayscale Class

```
namespace AForge.Imaging.Filters
{
    using System;
    using System.Collections.Generic;
    using System.Drawing;
    using System.Drawing.Imaging;
    /// <summary>
    /// Base class for image grayscaling.
    /// </summary>
    ///
    /// <remarks><para>This class is the base class for image grayscaling. Other
    /// classes should inherit from this class and specify <b>RGB</b>
    /// coefficients used for color image conversion to grayscale.</para>
    ///
    /// <para>The filter accepts 24, 32, 48 and 64 bpp color images and produces
    /// 8 (if source is 24 or 32 bpp image) or 16 (if source is 48 or 64 bpp image)
    /// bpp grayscale image.</para>
    ///
    /// <para>Sample usage:</para>
    /// <code>
    /// // create grayscale filter (BT709)
    /// Grayscale filter = new Grayscale( 0.2125, 0.7154, 0.0721 );
    /// // apply the filter
    /// Bitmap grayImage = filter.Apply( image );
    /// </code>
    ///
    /// <para><b>Initial image:</b></para>
    /// <img src="img/imaging/sample1.jpg" width="480" height="361" />
    /// <para><b>Result image:</b></para>
    /// <img src="img/imaging/grayscale.jpg" width="480" height="361" />
    /// </remarks>
    ///
    /// <seealso cref="GrayscaleBT709"/>
    /// <seealso cref="GrayscaleRMY"/>
    /// <seealso cref="GrayscaleY"/>
    ///
```

```
    public class Grayscale : BaseFilter
    {
        /// <summary>
        /// Set of predefined common grayscaling algorithms, which have aldready
initialized
        /// grayscaling coefficients.
        /// </summary>
        public static class CommonAlgorithms
        {
            /// <summary>
            /// Grayscale image using BT709 algorithm.
            /// </summary>
            ///
            /// <remarks><para>The instance uses <b>BT709</b> algorithm to convert
color image
            /// to grayscale. The conversion coefficients are:
            /// <list type="bullet">
            /// <item>Red: 0.2125;</item>
            /// <item>Green: 0.7154;</item>
            /// <item>Blue: 0.0721.</item>
            /// </list></para>
            ///
            /// <para>Sample usage:</para>
            /// <code>
            /// // apply the filter
            /// Bitmap grayImage = Grayscale.CommonAlgorithms.BT709.Apply( image
);
            /// </code>
            /// </remarks>
            ///
            public static readonly Grayscale BT709 = new Grayscale(0.2125, 0.7154,
0.0721);
            /// <summary>
            /// Grayscale image using R-Y algorithm.
            /// </summary>
            ///
            /// <remarks><para>The instance uses <b>R-Y</b> algorithm to convert
color image
            /// to grayscale. The conversion coefficients are:
            /// <list type="bullet">
            /// <item>Red: 0.5;</item>
            /// <item>Green: 0.419;</item>
```

```
        /// <item>Blue: 0.081.</item>
        /// </list></para>
        ///
        /// <para>Sample usage:</para>
        /// <code>
        /// // apply the filter
        /// Bitmap grayImage = Grayscale.CommonAlgorithms.RMY.Apply( image );
        /// </code>
        /// </remarks>
        ///
        public static readonly Grayscale RMY = new Grayscale(0.5000, 0.4190,
0.0810);
        /// <summary>
        /// Grayscale image using Y algorithm.
        /// </summary>
        ///
        /// <remarks><para>The instance uses <b>Y</b> algorithm to convert color
image
        /// to grayscale. The conversion coefficients are:
        /// <list type="bullet">
        /// <item>Red: 0.299;</item>
        /// <item>Green: 0.587;</item>
        /// <item>Blue: 0.114.</item>
        /// </list></para>
        ///
        /// <para>Sample usage:</para>
        /// <code>
        /// // apply the filter
        /// Bitmap grayImage = Grayscale.CommonAlgorithms.Y.Apply( image );
        /// </code>
        /// </remarks>
        ///
        public static readonly Grayscale Y = new Grayscale(0.2990, 0.5870, 0.1140);
    }
    // RGB coefficients for grayscale transformation
    /// <summary>
    /// Portion of red channel's value to use during conversion from RGB to
grayscale.
    /// </summary>
    public readonly double RedCoefficient;
    /// <summary>
```

```
        /// Portion of green channel's value to use during conversion from RGB to
grayscale.
        /// </summary>
        public readonly double GreenCoefficient;
        /// <summary>
        /// Portion of blue channel's value to use during conversion from RGB to
grayscale.
        /// </summary>
        public readonly double BlueCoefficient;
        // private format translation dictionary
        private Dictionary<PixelFormat, PixelFormat> formatTranslations = new
Dictionary<PixelFormat, PixelFormat>();
        /// <summary>
        /// Format translations dictionary.
        /// </summary>
        public override Dictionary<PixelFormat, PixelFormat> FormatTranslations
        {
            get { return formatTranslations; }
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Grayscale"/> class.
        /// </summary>
        ///
        /// <param name="cr">Red coefficient.</param>
        /// <param name="cg">Green coefficient.</param>
        /// <param name="cb">Blue coefficient.</param>
        ///
        public Grayscale(double cr, double cg, double cb)
        {
            RedCoefficient = cr;
            GreenCoefficient = cg;
            BlueCoefficient = cb;
            // initialize format translation dictionary
            formatTranslations[PixelFormat.Format24bppRgb] =
PixelFormat.Format8bppIndexed;
            formatTranslations[PixelFormat.Format32bppRgb] =
PixelFormat.Format8bppIndexed;
            formatTranslations[PixelFormat.Format32bppArgb] =
PixelFormat.Format8bppIndexed;
            formatTranslations[PixelFormat.Format48bppRgb] =
PixelFormat.Format16bppGrayScale;
```

```
            formatTranslations[PixelFormat.Format64bppArgb] =
PixelFormat.Format16bppGrayScale;
        }
        /// <summary>
        /// Process the filter on the specified image.
        /// </summary>
        ///
        /// <param name="sourceData">Source image data.</param>
        /// <param name="destinationData">Destination image data.</param>
        ///
        protected override unsafe void ProcessFilter(UnmanagedImage sourceData,
UnmanagedImage destinationData)
        {
            // get width and height
            int width = sourceData.Width;
            int height = sourceData.Height;
            PixelFormat srcPixelFormat = sourceData.PixelFormat;
            if (
            (srcPixelFormat == PixelFormat.Format24bppRgb) ||
            (srcPixelFormat == PixelFormat.Format32bppRgb) ||
            (srcPixelFormat == PixelFormat.Format32bppArgb))
            {
                int pixelSize = (srcPixelFormat == PixelFormat.Format24bppRgb) ? 3 : 4;
                int srcOffset = sourceData.Stride - width * pixelSize;
                int dstOffset = destinationData.Stride - width;
                int rc = (int)(0x10000 * RedCoefficient);
                int gc = (int)(0x10000 * GreenCoefficient);
                int bc = (int)(0x10000 * BlueCoefficient);
                // do the job
                byte* src = (byte*)sourceData.ImageData.ToPointer();
                byte* dst = (byte*)destinationData.ImageData.ToPointer();
                // for each line
                for (int y = 0; y < height; y++)
                {
                    // for each pixel
                    for (int x = 0; x < width; x++, src += pixelSize, dst++)
                    {
                        *dst = (byte)((rc * src[RGB.R] + gc * src[RGB.G] + bc * src[RGB.B])
>> 16);
                    }
                    src += srcOffset;
                    dst += dstOffset;
```

```
            }
        }
        else
        {
            int pixelSize = (srcPixelFormat == PixelFormat.Format48bppRgb) ? 3 : 4;
            byte* srcBase = (byte*)sourceData.ImageData.ToPointer();
            byte* dstBase = (byte*)destinationData.ImageData.ToPointer();
            int srcStride = sourceData.Stride;
            int dstStride = destinationData.Stride;
            // for each line
            for (int y = 0; y < height; y++)
            {
                ushort* src = (ushort*)(srcBase + y * srcStride);
                ushort* dst = (ushort*)(dstBase + y * dstStride);
                // for each pixel
                for (int x = 0; x < width; x++, src += pixelSize, dst++)
                {
                    *dst = (ushort)(RedCoefficient * src[RGB.R] + GreenCoefficient *
src[RGB.G] + BlueCoefficient * src[RGB.B]);
                }
            }
        }
    }
}
```

A.2 Difference Class

```
namespace AForge.Imaging.Filters
{
    using System;
    using System.Collections.Generic;
    using System.Drawing;
    using System.Drawing.Imaging;
    /// <summary>
    /// Difference filter - get the difference between overlay and source images.
    /// </summary>
    ///
    /// <remarks><para>The difference filter takes two images (source and
    /// <see cref="BaseInPlaceFilter2.OverlayImage">overlay</see> images)
```

```
    /// of the same size and pixel format and produces an image, where each pixel
equals
    /// to absolute difference between corresponding pixels from provided
images.</para>
    ///
    /// <para>The filter accepts 8 and 16 bpp grayscale images and 24, 32, 48 and 64
bpp
    /// color images for processing.</para>
    ///
    /// <para><note>In the case if images with alpha channel are used (32 or 64 bpp),
visualization
    /// of the result image may seem a bit unexpected - most probably nothing will be
seen
    /// (in the case if image is displayed according to its alpha channel). This may be
    /// caused by the fact that after differencing the entire alpha channel will be zeroed
    /// (zero difference between alpha channels), what means that the resulting image
will be
    /// 100% transparent.</note></para>
    ///
    /// <para>Sample usage:</para>
    /// <code>
    ///  // create filter
    /// Difference filter = new Difference( overlayImage );
    /// // apply the filter
    /// Bitmap resultImage = filter.Apply( sourceImage );
    /// </code>
    ///
    /// <para><b>Source image:</b></para>
    /// <img src="img/imaging/sample6.png" width="320" height="240" />
    /// <para><b>Overlay image:</b></para>
    /// <img src="img/imaging/sample7.png" width="320" height="240" />
    /// <para><b>Result image:</b></para> /// <img
src="img/imaging/difference.png" width="320" height="240" />
    /// </remarks>
    ///
    /// <seealso cref="Intersect"/>
    /// <seealso cref="Merge"/>
    /// <seealso cref="Add"/>
    /// <seealso cref="Subtract"/>
    ///
    public sealed class Difference : BaseInPlaceFilter2
    {
```

```
        // private format translation dictionary private Dictionary<PixelFormat,
PixelFormat> formatTranslations = new Dictionary<PixelFormat, PixelFormat>( );
        /// <summary>
        /// Format translations dictionary.
        /// </summary>
        public override Dictionary<PixelFormat, PixelFormat> FormatTranslations
        {
            get { return formatTranslations; }
        }
        /// <summary>
        ///Initializes a new instance of the <see cref="Difference"/> class.
        /// </summary>
        public Difference()
        {
            InitFormatTranslations();
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Difference"/> class.
        /// </summary>
        ///
        /// <param name="overlayImage">Overlay image.</param>
        ///
        public Difference(Bitmap overlayImage)
            : base(overlayImage)
        {
            InitFormatTranslations();
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Difference"/> class.
        /// </summary>
        ///
        /// <param name="unmanagedOverlayImage">Unmanaged overlay
image.</param> ///
        public Difference(UnmanagedImage unmanagedOverlayImage)
            : base(unmanagedOverlayImage)
        {
            InitFormatTranslations();
        }
        // Initialize format translation dictionary
        private void InitFormatTranslations()
        {
```

```
            formatTranslations[PixelFormat.Format8bppIndexed] =
PixelFormat.Format8bppIndexed;
            formatTranslations[PixelFormat.Format24bppRgb] =
PixelFormat.Format24bppRgb;
            formatTranslations[PixelFormat.Format32bppRgb] =
PixelFormat.Format32bppRgb;
            formatTranslations[PixelFormat.Format32bppArgb] =
PixelFormat.Format32bppArgb;
            formatTranslations[PixelFormat.Format16bppGrayScale] =
PixelFormat.Format16bppGrayScale;
            formatTranslations[PixelFormat.Format48bppRgb] =
PixelFormat.Format48bppRgb;
            formatTranslations[PixelFormat.Format64bppArgb] =
PixelFormat.Format64bppArgb;
        }
        /// <summary>
        /// Process the filter on the specified image.
        /// </summary>
        ///
        /// <param name="image">Source image data.</param>
        /// <param name="overlay">Overlay image data.</param>
        ///
        protected override unsafe void ProcessFilter(UnmanagedImage image,
UnmanagedImage overlay)
        {
            PixelFormat pixelFormat = image.PixelFormat;
            // get image dimension
            int width = image.Width;
            int height = image.Height;
            // pixel value
            int v;
            if (
            (pixelFormat == PixelFormat.Format8bppIndexed) ||
            (pixelFormat == PixelFormat.Format24bppRgb) ||
            (pixelFormat == PixelFormat.Format32bppRgb) ||
            (pixelFormat == PixelFormat.Format32bppArgb))
            {
                // initialize other variables
                int pixelSize = (pixelFormat == PixelFormat.Format8bppIndexed) ? 1 :
                (pixelFormat == PixelFormat.Format24bppRgb) ? 3 : 4;
                int lineSize = width * pixelSize;
                int srcOffset = image.Stride - lineSize;
```

```
    int ovrOffset = overlay.Stride - lineSize;
    // do the job
    byte* ptr = (byte*)image.ImageData.ToPointer();
    byte* ovr = (byte*)overlay.ImageData.ToPointer();
    // for each line
    for (int y = 0; y < height; y++)
    {
        // for each pixel
        for (int x = 0; x < lineSize; x++, ptr++, ovr++)
        {
            // abs(sub)
            v = (int)*ptr - (int)*ovr;
            *ptr = (v < 0) ? (byte)-v : (byte)v;
        }
        ptr += srcOffset;
        ovr += ovrOffset;
    }
}
else
{
    // initialize other variables
    int pixelSize = (pixelFormat == PixelFormat.Format16bppGrayScale) ? 1 :
    (pixelFormat == PixelFormat.Format48bppRgb) ? 3 : 4;
    int lineSize = width * pixelSize;
    int srcStride = image.Stride;
    int ovrStride = overlay.Stride;
    // do the job
    int basePtr = (int)image.ImageData.ToPointer();
    int baseOvr = (int)overlay.ImageData.ToPointer();
    // for each line
    for (int y = 0; y < height; y++)
    {
        ushort* ptr = (ushort*)(basePtr + y * srcStride);
        ushort* ovr = (ushort*)(baseOvr + y * ovrStride);
        // for each pixel
        for (int x = 0; x < lineSize; x++, ptr++, ovr++)
        {
            // abs(sub)
            v = (int)*ptr - (int)*ovr;
            *ptr = (v < 0) ? (ushort)-v : (ushort)v;
        }
    }
```

```
        }
      }
    }
}
```

A.3 Threshold Class

```
namespace AForge.Imaging.Filters
{
    using System;
    using System.Collections.Generic;
    using System.Drawing;
    using System.Drawing.Imaging;
    /// <summary>
    /// Threshold binarization.
    /// </summary>
    ///
    /// <remarks><para>The filter does image binarization using specified threshold
value. All pixels
    /// with intensities equal or higher than threshold value are converted to white
pixels. All other
    /// pixels with intensities below threshold value are converted to black
pixels.</para>
    ///
    /// <para>The filter accepts 8 and 16 bpp grayscale images for processing.</para>
    ///
    /// <para><note>Since the filter can be applied as to 8 bpp and to 16 bpp images,
    /// the <see cref="ThresholdValue"/> value should be set appropriately to the pixel
format.
    /// In the case of 8 bpp images the threshold value is in the [0, 255] range, but in
the case
    /// of 16 bpp images the threshold value is in the [0, 65535] range.</note></para>
    ///
    /// <para>Sample usage:</para>
    /// <code>
    /// // create filter
    /// Threshold filter = new Threshold( 100 );
    /// // apply the filter
    /// filter.ApplyInPlace( image );
    /// </code>
    ///
    /// <para><b>Initial image:</b></para>
    /// <img src="img/imaging/grayscale.jpg" width="480" height="361" />
```

```
    /// <para><b>Result image:</b></para>
    /// <img src="img/imaging/threshold.jpg" width="480" height="361" />
    /// </remarks>
    ///
    public class Threshold : BaseInPlacePartialFilter
    {
        /// <summary>
        /// Threshold value.
        /// </summary>
        protected int threshold = 128;
        // private format translation dictionary
        private Dictionary<PixelFormat, PixelFormat> formatTranslations = new
Dictionary<PixelFormat, PixelFormat>();
        /// <summary>
        /// Format translations dictionary.
        /// </summary>
        public override Dictionary<PixelFormat, PixelFormat> FormatTranslations
        {
            get { return formatTranslations; }
        }
        /// <summary>
        /// Threshold value.
        /// </summary>
        ///
        /// <remarks>Default value is set to <b>128</b>.</remarks>
        ///
        public int ThresholdValue
        {
            get { return threshold; }
            set { threshold = value; }
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Threshold"/> class.
        /// </summary>
        ///
        public Threshold()
        {
            // initialize format translation dictionary
            formatTranslations[PixelFormat.Format8bppIndexed] =
PixelFormat.Format8bppIndexed;
            formatTranslations[PixelFormat.Format16bppGrayScale] =
PixelFormat.Format16bppGrayScale;
```

```
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Threshold"/> class.
        /// </summary>
        ///
        /// <param name="threshold">Threshold value.</param>
        ///
        public Threshold(int threshold)
            : this()
        {
            this.threshold = threshold;
        }
        /// <summary>
        /// Process the filter on the specified image.
        /// </summary>
        ///
        /// <param name="image">Source image data.</param>
        /// <param name="rect">Image rectangle for processing by the filter.</param>
        ///
        protected override unsafe void ProcessFilter(UnmanagedImage image,
Rectangle rect)
        {
            int startX = rect.Left;
            int startY = rect.Top;
            int stopX = startX + rect.Width;
            int stopY = startY + rect.Height;
            if (image.PixelFormat == PixelFormat.Format8bppIndexed)
            {
                int offset = image.Stride - rect.Width;
                // do the job
                byte* ptr = (byte*)image.ImageData.ToPointer();
                // allign pointer to the first pixel to process
                ptr += (startY * image.Stride + startX);
                // for each line
                for (int y = startY; y < stopY; y++)
                {
                    // for each pixel
                    for (int x = startX; x < stopX; x++, ptr++)
                    {
                        *ptr = (byte)((*ptr >= threshold) ? 255 : 0);
                    }
                    ptr += offset;
```

```
            }
        }
        else
        {
            byte* basePtr = (byte*)image.ImageData.ToPointer() + startX * 2;
            int stride = image.Stride;
            // for each line
            for (int y = startY; y < stopY; y++)
            {
                ushort* ptr = (ushort*)(basePtr + stride * y);
                // for each pixel
                for (int x = startX; x < stopX; x++, ptr++)
                {
                    *ptr = (ushort)((*ptr >= threshold) ? 65535 : 0);
                }
            }
        }
    }
  }
}
```

A.4 Erosion Class

```
namespace AForge.Imaging.Filters
{
  using System;
  using System.Collections.Generic;
  using System.Drawing;
  using System.Drawing.Imaging;
  /// <summary>
  /// Erosion operator from Mathematical Morphology.
  /// </summary>
  ///
  /// <remarks><para>The filter assigns minimum value of surrounding pixels to
each pixel of
  /// the result image. Surrounding pixels, which should be processed, are specified
by
  /// structuring element: 1 - to process the neighbor, -1 - to skip it.</para>
  ///
  /// <para>The filter especially useful for binary image processing, where it
removes pixels, which
```

```
    /// are not surrounded by specified amount of neighbors. It gives ability to remove noisy pixels
    /// (stand-alone pixels) or shrink objects.</para>
    ///
    /// <para>For processing image with 3x3 structuring element, there are different optimizations
    /// available, like <see cref="Erosion3x3"/> and <see cref="BinaryErosion3x3"/>.</para>
    ///
    /// <para>The filter accepts 8 and 16 bpp grayscale images and 24 and 48 bpp
    /// color images for processing.</para>
    ///
    /// <para>Sample usage:</para>
    /// <code>
    /// // create filter
    /// Erosion filter = new Erosion( );
    /// // apply the filter
    /// filter.Apply( image );
    /// </code>
    ///
    /// <para><b>Initial image:</b></para>
    /// <img src="img/imaging/sample12.png" width="320" height="240" />
    /// <para><b>Result image:</b></para>
    /// <img src="img/imaging/erosion.png" width="320" height="240" />
    /// </remarks>
    ///
    /// <seealso cref="Dilatation"/>
    /// <seealso cref="Closing"/>
    /// <seealso cref="Opening"/>
    /// <seealso cref="Erosion3x3"/>
    /// <seealso cref="BinaryErosion3x3"/>
    ///
    public class Erosion : BaseUsingCopyPartialFilter
    {
        // structuring element
        private short[,] se = new short[3, 3] { { 1, 1, 1 }, { 1, 1, 1 }, { 1, 1, 1 } };
        private int size = 3;
        // private format translation dictionary
        private Dictionary<PixelFormat, PixelFormat> formatTransalations = new Dictionary<PixelFormat, PixelFormat>();
        /// <summary>
        /// Format translations dictionary.
```

```
        /// </summary>
        public override Dictionary<PixelFormat, PixelFormat> FormatTransalations
        {
            get { return formatTransalations; }
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Erosion"/> class.
        /// </summary>
        ///
        /// <remarks><para>Initializes new instance of the <see cref="Erosion"/> class
using
        /// default structuring element - 3x3 structuring element with all elements equal
to 1.
        /// </para></remarks>
        ///
        public Erosion()
        {
            // initialize format translation dictionary
            formatTransalations[PixelFormat.Format8bppIndexed] =
PixelFormat.Format8bppIndexed;
            formatTransalations[PixelFormat.Format24bppRgb] =
PixelFormat.Format24bppRgb;
            formatTransalations[PixelFormat.Format16bppGrayScale] =
PixelFormat.Format16bppGrayScale;
            formatTransalations[PixelFormat.Format48bppRgb] =
PixelFormat.Format48bppRgb;
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="Erosion"/> class.
        /// </summary>
        ///
        /// <param name="se">Structuring element.</param>
        ///
        /// <remarks><para>Structuring elemement for the erosion morphological
operator
        /// must be square matrix with odd size in the range of [3,
99].</para></remarks>
        ///
        /// <exception cref="ArgumentException">Invalid size of structuring
element.</exception>
        ///
        public Erosion(short[,] se)
```

```
        : this()
    {
        int s = se.GetLength(0);
        // check structuring element size
        if ((s != se.GetLength(1)) || (s < 3) || (s > 99) || (s % 2 == 0))
            throw new ArgumentException("Invalid size of structuring element.");
        this.se = se;
        this.size = s;
    }
    /// <summary>
    /// Process the filter on the specified image.
    /// </summary>
    ///
    /// <param name="sourceData">Source image data.</param>
    /// <param name="destinationData">Destination image data.</param>
    /// <param name="rect">Image rectangle for processing by the filter.</param>
    ///
    protected override unsafe void ProcessFilter(UnmanagedImage sourceData,
UnmanagedImage destinationData, Rectangle rect)
    {
        PixelFormat pixelFormat = sourceData.PixelFormat;
        // processing start and stop X,Y positions
        int startX = rect.Left;
        int startY = rect.Top;
        int stopX = startX + rect.Width;
        int stopY = startY + rect.Height;
        // structuring element's radius
        int r = size >> 1;
        if ((pixelFormat == PixelFormat.Format8bppIndexed) || (pixelFormat ==
PixelFormat.Format24bppRgb))
        {
            int pixelSize = (pixelFormat == PixelFormat.Format8bppIndexed) ? 1 : 3;
            int dstStride = destinationData.Stride;
            int srcStride = sourceData.Stride;
            // base pointers
            byte* baseSrc = (byte*)sourceData.ImageData.ToPointer();
            byte* baseDst = (byte*)destinationData.ImageData.ToPointer();
            // allign pointers by X
            baseSrc += (startX * pixelSize);
            baseDst += (startX * pixelSize);
            if (pixelFormat == PixelFormat.Format8bppIndexed)
            {
```

```
// grayscale image
// compute each line
for (int y = startY; y < stopY; y++)
{
   byte* src = baseSrc + y * srcStride;
   byte* dst = baseDst + y * dstStride;
   byte min, v;
   // loop and array indexes
   int t, ir, jr, i, j;
   // for each pixel
   for (int x = startX; x < stopX; x++, src++, dst++)
   {
      min = 255;
      // for each structuring element's row
      for (i = 0; i < size; i++)
      {
         ir = i - r;
         t = y + ir;
         // skip row
         if (t < startY)
            continue;
         // break
         if (t >= stopY)
            break;
         // for each structuring element's column
         for (j = 0; j < size; j++)
         {
            jr = j - r;
            t = x + jr;
            // skip column
            if (t < startX)
               continue;
            if (t < stopX)
            {
               if (se[i, j] == 1)
               {
                  // get new MIN value
                  v = src[ir * srcStride + jr];
                  if (v < min)
                     min = v;
               }
            }
```

```
            }
          }
          // result pixel
          *dst = min;
        }
      }
    }
    else
    {
      // 24 bpp color image
      // compute each line
      for (int y = startY; y < stopY; y++)
      {
        byte* src = baseSrc + y * srcStride;
        byte* dst = baseDst + y * dstStride;
        byte minR, minG, minB, v;
        byte* p;
        // loop and array indexes
        int t, ir, jr, i, j;
        // for each pixel
        for (int x = startX; x < stopX; x++, src += 3, dst += 3)
        {
          minR = minG = minB = 255;
          // for each structuring element's row
          for (i = 0; i < size; i++)
          {
            ir = i - r;
            t = y + ir;
            // skip row
            if (t < startY)
              continue;
            // break
            if (t >= stopY)
              break;
            // for each structuring element's column
            for (j = 0; j < size; j++)
            {
              jr = j - r;
              t = x + jr;
              // skip column
              if (t < startX)
                continue;
```

```
                        if (t < stopX)
                        {
                            if (se[i, j] == 1)
                            {
                                // get new MIN values
                                p = &src[ir * srcStride + jr * 3];
                                // red
                                v = p[RGB.R];
                                if (v < minR)
                                    minR = v;
                                // green
                                v = p[RGB.G];
                                if (v < minG)
                                    minG = v;
                                // blue
                                v = p[RGB.B];
                                if (v < minB)
                                    minB = v;
                            }
                        }
                    }
                }
                // result pixel
                dst[RGB.R] = minR;
                dst[RGB.G] = minG;
                dst[RGB.B] = minB;
            }
        }
    }
}
else
{
    int pixelSize = (pixelFormat == PixelFormat.Format16bppGrayScale) ? 1 :
3;
    int dstStride = destinationData.Stride / 2;
    int srcStride = sourceData.Stride / 2;
    // base pointers
    ushort* baseSrc = (ushort*)sourceData.ImageData.ToPointer();
    ushort* baseDst = (ushort*)destinationData.ImageData.ToPointer();
    // allign pointers by X
    baseSrc += (startX * pixelSize);
    baseDst += (startX * pixelSize);
```

```
if (pixelFormat == PixelFormat.Format16bppGrayScale)
{
   // 16 bpp grayscale image
   // compute each line
   for (int y = startY; y < stopY; y++)
   {
      ushort* src = baseSrc + y * srcStride;
      ushort* dst = baseDst + y * dstStride;
      ushort min, v;
      // loop and array indexes
      int t, ir, jr, i, j;
      // for each pixel
      for (int x = startX; x < stopX; x++, src++, dst++)
      {
         min = 65535;
         // for each structuring element's row
         for (i = 0; i < size; i++)
         {
            ir = i - r;
            t = y + ir;
            // skip row
            if (t < startY)
               continue;
            // break
            if (t >= stopY)
               break;
            // for each structuring element's column
            for (j = 0; j < size; j++)
            {
               jr = j - r;
               t = x + jr;
               // skip column
               if (t < startX)
                  continue;
               if (t < stopX)
               {
                  if (se[i, j] == 1)
                  {
                     // get new MIN value
                     v = src[ir * srcStride + jr];
                     if (v < min)
                        min = v;
```

```
                    }
                }
            }
        }
        // result pixel
        *dst = min;
    }
  }
}
else
{
  // 48 bpp color image
  // compute each line
  for (int y = startY; y < stopY; y++)
  {
    ushort* src = baseSrc + y * srcStride;
    ushort* dst = baseDst + y * dstStride;
    ushort minR, minG, minB, v;
    ushort* p;
    // loop and array indexes
    int t, ir, jr, i, j;
    // for each pixel
    for (int x = startX; x < stopX; x++, src += 3, dst += 3)
    {
      minR = minG = minB = 65535;
      // for each structuring element's row
      for (i = 0; i < size; i++)
      {
        ir = i - r;
        t = y + ir;
        // skip row
        if (t < startY)
          continue;
        // break
        if (t >= stopY)
          break;
        // for each structuring element's column
        for (j = 0; j < size; j++)
        {
          jr = j - r;
          t = x + jr;
          // skip column
```

```
                        if (t < startX)
                            continue;
                        if (t < stopX)
                        {
                            if (se[i, j] == 1)
                            {
                                // get new MIN values
                                p = &src[ir * srcStride + jr * 3];
                                // red
                                v = p[RGB.R];
                                if (v < minR)
                                    minR = v;
                                // green
                                v = p[RGB.G];
                                if (v < minG)
                                    minG = v;
                                // blue
                                v = p[RGB.B];
                                if (v < minB)
                                    minB = v;
                            }
                        }
                    }
                }
                // result pixel
                dst[RGB.R] = minR;
                dst[RGB.G] = minG;
                dst[RGB.B] = minB;
            }
        }
    }
  }
 }
}
```

LAMPIRAN B

LIST PROGRAM MOTION DETECTION WITH WATER EFFECT

B.1 Camera Window

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

namespace Gerakan_Efek_Air
{
   public partial class CameraWindow : Control
   {
      private Camera camera = null;
      private bool autosize = false;
      private bool needSizeUpdate = false;
      private bool firstFrame = true;

      private Color rectColor = Color.Black;


      //AutoSize property
      [DefaultValue(false)]
      public bool AutoSize
      {
         get { return autosize; }
         set
         {
            autosize = value;
            UpdatePosition();
         }
      }

      // Camera property
      [Browsable(false)]
      public Camera Camera
      {
         get { return camera; }
         set
```

```
    {
        // lock
        Monitor.Enter(this);

        // detach event
        if (camera != null)
        {
            camera.NewFrame -= new EventHandler(camera_NewFrame);
            timer.Stop();
        }

        camera = value;
        needSizeUpdate = true;
        firstFrame = true;

        // atach event
        if (camera != null)
        {
            camera.NewFrame += new EventHandler(camera_NewFrame);
            timer.Start();
        }

        // unlock
        Monitor.Exit(this);
    }
}

public System.Drawing.Bitmap bmp
{
    get { return bmp; }
    set { bmp = value; }
}

public CameraWindow()
{
    InitializeComponent();
    SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.DoubleBuffer |
      ControlStyles.ResizeRedraw | ControlStyles.UserPaint, true);
}

public CameraWindow(IContainer container)
{
```

```
            container.Add(this);
            InitializeComponent();
        }

        // Paint control
        protected override void OnPaint(PaintEventArgs pe)
        {
            if ((needSizeUpdate) || (firstFrame))
            {
                UpdatePosition();
                needSizeUpdate = false;
            }

            // lock
            Monitor.Enter(this);

            Graphics g = pe.Graphics;
            Rectangle rc = this.ClientRectangle;
            Pen pen = new Pen(rectColor, 1);

            // draw rectangle
            g.DrawRectangle(pen, rc.X, rc.Y, rc.Width - 1, rc.Height - 1);

            if (camera != null)
            {
                try
                {
                    camera.Lock();

                    // draw frame
                    if (camera.LastFrameLocal != null)
                    {
                        g.DrawImage(camera.LastFrameLocal, rc.X + 1, rc.Y + 1, rc.Width -
2, rc.Height - 2);
                        firstFrame = false;
                    }
                    else
                    {
                        // Create font and brush
                        Font drawFont = new Font("Arial", 12);
                        SolidBrush drawBrush = new SolidBrush(Color.White);
```

```
                    g.DrawString("Connecting ...", drawFont, drawBrush, new PointF(5,
5));

                    drawBrush.Dispose();
                    drawFont.Dispose();
                }
            }
            catch (Exception)
            {
            }
            finally
            {
                camera.Unlock();
            }
        }

        pen.Dispose();

        // unlock
        Monitor.Exit(this);

        base.OnPaint(pe);
    }

    // Update position and size of the control
    public void UpdatePosition()
    {
        // lock
        Monitor.Enter(this);

        if ((autosize) && (this.Parent != null))
        {
            Rectangle rc = this.Parent.ClientRectangle;
            int width = 320;
            int height = 240;

            if (camera != null)
            {
                camera.Lock();

                // get frame dimension
                if (camera.LastFrame != null)
```

```
                {
                    width = camera.LastFrameLocal.Width;
                    height = camera.LastFrameLocal.Height;
                }
                camera.Unlock();
            }

            //
            this.SuspendLayout();
            this.Location = new Point((rc.Width - width - 2) / 2, (rc.Height - height - 2)
/ 2);
            this.Size = new Size(width + 2, height + 2);
            this.ResumeLayout();

        }
        // unlock
        Monitor.Exit(this);
    }

    // On new frame ready
    private void camera_NewFrame(object sender, System.EventArgs e)
    {
        Invalidate();
    }
  }
}
```

B.2 Camera

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Threading;

using VideoSource;

using AForge.Imaging;
using AForge.Imaging.Filters;

namespace Gerakan_Efek_Air
{
```

```
public class Camera
{
    private IVideoSource videoSource = null;
    private CaptureDevice localSource = null;

    private Bitmap lastFrameLocal = null;

    private Bitmap lastFrame = null;

    //private proses image
    private IFilter grayscaleFilter = new GrayscaleBT709();
    private Difference differenceFilter = new Difference();
    private Threshold thresholdFilter = new Threshold(15);
    private IFilter erosionFilter = new Erosion();
    private Merge mergeFilter = new Merge();
    private IFilter extrachChannel = new ExtractChannel(RGB.R);
    private ReplaceChannel replaceChannel = new ReplaceChannel(RGB.R, null);

    // image width and height
    private int width = -1, height = -1;

    private Bitmap backgroundFrame = null;
    private BitmapData bitmapData;

    public event EventHandler NewFrame;

    // LastFrame property
    public Bitmap LastFrame
    {
        get { return lastFrame; }
    }

    // Width property
    public int Width
    {
        get { return width; }
    }
    // Height property
    public int Height
    {
        get { return height; }
    }
```

```
// Running property
public bool Running
{
   get { return (localSource == null) ? false : localSource.Running; }
}

// LastFrameLocal property
public Bitmap LastFrameLocal
{
   get { return lastFrameLocal; }
}

//Constructor
public Camera(CaptureDevice localSource)
{
   this.localSource = localSource;
   localSource.NewFrame += new CameraEventHandler(video_NewFrame);
}


//Start video Source
public void Start()
{
   if (videoSource != null)
   {
      videoSource.Start();
   }
   if (localSource != null)
   {
      localSource.Start();
   }

}

//Signal Video Source to Stop
public void SignalToStop()
{
   if (videoSource != null)
   {
      videoSource.SignalToStop();
   }
   if (localSource != null)
```

```
    {
        localSource.SignalToStop();
    }
}

//wait video source to Stop
public void WaitForStop()
{
    if (videoSource != null)
    {
        videoSource.WaitForStop();
    }
    if (localSource != null)
    {
        localSource.WaitForStop();
    }
}

//Abort Camera
public void Stop()
{
    if (videoSource != null)
    {
        videoSource.Stop();
    }
    if (localSource != null)
    {
        localSource.Stop();
    }
}



// Lock it
public void Lock()
{
    Monitor.Enter(this);
}

// Unlock it
public void Unlock()
{
```

```
    Monitor.Exit(this);
}

private void video_NewFrame(object sender, CameraEventArgs e)
{
    try
    {
        //lock
        Monitor.Enter(this);

        //dispose old frame
        if (lastFrame != null)
        {
            lastFrame.Dispose();
        }

        lastFrame = (Bitmap)e.Bitmap.Clone();

        if (backgroundFrame == null)
        {
            // create initial backgroung image
            backgroundFrame = grayscaleFilter.Apply(lastFrame);

            // get image dimension
            width = lastFrame.Width;
            height = lastFrame.Height;

            // just return for the first time
            return;
        }

        Bitmap tmpImage;

        // apply the grayscale file
        tmpImage = grayscaleFilter.Apply(lastFrame);

        // set backgroud frame as an overlay for difference filter
        differenceFilter.OverlayImage = backgroundFrame;

        // apply difference filter
        Bitmap tmpImage2 = differenceFilter.Apply(tmpImage);
```

```
            // lock the temporary image and apply some filters on the locked data
            bitmapData = tmpImage2.LockBits(new Rectangle(0, 0, width, height),
                ImageLockMode.ReadWrite, PixelFormat.Format8bppIndexed);

            // threshold filter
            thresholdFilter.ApplyInPlace(bitmapData);
            // erosion filter
            Bitmap tmpImage3 = erosionFilter.Apply(bitmapData);

            Bitmap tmpImage4 = tmpImage3.Clone(new Rectangle(0, 0,
tmpImage3.Width, tmpImage3.Height), PixelFormat.Format24bppRgb);

            // unlock temporary image
            tmpImage2.UnlockBits(bitmapData);
            tmpImage2.Dispose();

            // dispose old background
            backgroundFrame.Dispose();
            // set backgound to current
            backgroundFrame = tmpImage;


            lastFrameLocal = tmpImage4;


        }
        catch (Exception)
        {
        }
        finally
        {
            //unlock
            Monitor.Exit(this);
        }


        // notify client
        if (NewFrame != null)
            NewFrame(this, new EventArgs());
    }
  }
}
```

B.3 Capture Device Form

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Threading;

using VideoSource;

using AForge.Imaging;
using AForge.Imaging.Filters;

namespace Gerakan_Efek_Air
{
   public class Camera
   {
      private IVideoSource videoSource = null;
      private CaptureDevice localSource = null;

      private Bitmap lastFrameLocal = null;

      private Bitmap lastFrame = null;

      //private proses image
      private IFilter grayscaleFilter = new GrayscaleBT709();
      private Difference differenceFilter = new Difference();
      private Threshold thresholdFilter = new Threshold(15);
      private IFilter erosionFilter = new Erosion();
      private Merge mergeFilter = new Merge();
      private IFilter extrachChannel = new ExtractChannel(RGB.R);
      private ReplaceChannel replaceChannel = new ReplaceChannel(RGB.R, null);

      // image width and height
      private int width = -1, height = -1;

      private Bitmap backgroundFrame = null;
      private BitmapData bitmapData;

      public event EventHandler NewFrame;
```

```
// LastFrame property
public Bitmap LastFrame
{
   get { return lastFrame; }
}

// Width property
public int Width
{
   get { return width; }
}
// Height property
public int Height
{
   get { return height; }
}
// Running property
public bool Running
{
   get { return (localSource == null) ? false : localSource.Running; }
}

// LastFrameLocal property
public Bitmap LastFrameLocal
{
   get { return lastFrameLocal; }
}

//Constructor
public Camera(CaptureDevice localSource)
{
   this.localSource = localSource;
   localSource.NewFrame += new CameraEventHandler(video_NewFrame);
}


//Start video Source
public void Start()
{
   if (videoSource != null)
   {
      videoSource.Start();
```

```
    }
    if (localSource != null)
    {
        localSource.Start();
    }

}

//Signal Video Source to Stop
public void SignalToStop()
{
    if (videoSource != null)
    {
        videoSource.SignalToStop();
    }
    if (localSource != null)
    {
        localSource.SignalToStop();
    }
}

//wait video source to Stop
public void WaitForStop()
{
    if (videoSource != null)
    {
        videoSource.WaitForStop();
    }
    if (localSource != null)
    {
        localSource.WaitForStop();
    }
}

//Abort Camera
public void Stop()
{
    if (videoSource != null)
    {
        videoSource.Stop();
    }
    if (localSource != null)
```

```
    {
        localSource.Stop();
    }
}



// Lock it
public void Lock()
{
    Monitor.Enter(this);
}

// Unlock it
public void Unlock()
{
    Monitor.Exit(this);
}

private void video_NewFrame(object sender, CameraEventArgs e)
{
    try
    {
        //lock
        Monitor.Enter(this);

        //dispose old frame
        if (lastFrame != null)
        {
            lastFrame.Dispose();
        }

        lastFrame = (Bitmap)e.Bitmap.Clone();

        if (backgroundFrame == null)
        {
            // create initial backgroung image
            backgroundFrame = grayscaleFilter.Apply(lastFrame);

            // get image dimension
            width = lastFrame.Width;
            height = lastFrame.Height;
```

```
            // just return for the first time
            return;
        }

        Bitmap tmpImage;

        // apply the grayscale file
        tmpImage = grayscaleFilter.Apply(lastFrame);

        // set backgroud frame as an overlay for difference filter
        differenceFilter.OverlayImage = backgroundFrame;

        // apply difference filter
        Bitmap tmpImage2 = differenceFilter.Apply(tmpImage);

        // lock the temporary image and apply some filters on the locked data
        bitmapData = tmpImage2.LockBits(new Rectangle(0, 0, width, height),
            ImageLockMode.ReadWrite, PixelFormat.Format8bppIndexed);

        // threshold filter
        thresholdFilter.ApplyInPlace(bitmapData);
        // erosion filter
        Bitmap tmpImage3 = erosionFilter.Apply(bitmapData);

        Bitmap tmpImage4 = tmpImage3.Clone(new Rectangle(0, 0,
tmpImage3.Width, tmpImage3.Height), PixelFormat.Format24bppRgb);

        // unlock temporary image
        tmpImage2.UnlockBits(bitmapData);
        tmpImage2.Dispose();

        // dispose old background
        backgroundFrame.Dispose();
        // set backgound to current
        backgroundFrame = tmpImage;


        lastFrameLocal = tmpImage4;


    }
```

```
            catch (Exception)
            {
            }
            finally
            {
                //unlock
                Monitor.Exit(this);
            }


            // notify client
            if (NewFrame != null)
                NewFrame(this, new EventArgs());
        }
    }
}
```

B.4 Gerakan Efek Air

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Collections;

using VideoSource;


namespace Gerakan_Efek_Air
{
    public partial class FormEfek : Form
    {
        //private webcam
        private static CaptureDevice localSource = new CaptureDevice();

        //inisialisai Webcam
        Camera camera = new Camera(localSource);
```

```
//inisialisai gambar
private Bitmap image1;
private Bitmap OriginalImage;

//Inisialisai posisi Mouse
public static int pointX;
public static int pointY;

//Inisialisasi warna putih
private Color cWhite = System.Drawing.Color.FromArgb(255, 255, 255, 255);

//Inisialisasi FormState
FormState formstate = new FormState();

//Constructor
public FormEfek()
{
   InitializeComponent();
}

//Form Gerakan Efek Air Load
private void FormEfek_Load(object sender, System.EventArgs e)
{
   //Menampilkan Capture Device Form
   CaptureDeviceForm form = new CaptureDeviceForm();
   if (form.ShowDialog(this) == DialogResult.OK)
   {
      localSource.VideoSource = form.Device;  //Camera dipilih
      this.Cursor = Cursors.WaitCursor;

      //Memulai Camera
      camera.Start();

      this.Cursor = Cursors.Default;
   }

   if (camera != null)
   {
      //timerImage dijalankan untuk mengambil frame
      timerImage.Start();
   }
}
```

```
//TimerImage jalan, frame mulai diambil
private void timerImage_Tick(object sender, System.EventArgs e)
{
   image1 = camera.LastFrameLocal;
   //timerVertex dijalankan untuk mengolah frame
   timerVertex.Start();
}

//TimerVertex diambil untuk mengolah frame
private void timerVertex_Tick(object sender, System.EventArgs e)
{
   if (image1 != null)  // Jika frame berhasil didapat
   {
      labelMotion.Text = " Motion Detection Ok";
      OriginalImage = (Bitmap)image1.Clone();
      ArrayList pxList = scan(OriginalImage); //frame yg didapat di scan

      //Pixel Putih dihitung
      int a = pxList.Count;
      int[] arrayX = new int[a];
      int[] arrayY = new int[a];

      //Mengenali tiap titik yg didapat
      Vertex prv;
      for (int k = 0; k < pxList.Count; k++)
      {
         prv = (Vertex)pxList[k];
         arrayX[k] = prv.X;
         arrayY[k] = prv.Y;
      }

      //Menentukan Banyak titik yg didapat
      int countX = arrayX.Length;
      int countY = arrayY.Length;

      //Mengenali gerakan untuk menentukan posisi mouse
      if (countX != 0)
      {
         //posisi koordinat X ditentukan
         pointX = Convert.ToInt32(arrayX[countX / 2]);
      }
      else
```

```
            {
                pointX = 0;
            }
            labelPointX.Text = "pointX = " + pointX*2 + ""; // label menampilkan
posisi X kursor mouse

            if (countY != 0)
            {
                //posisi koordinat Y ditentukan
                pointY = Convert.ToInt32(arrayY[countY / 2]);
            }
            else
            {
                pointY = 0;
            }
            labelPointY.Text = "pointY = " + pointY*2 + "";// label menampilkan
posisi Y kursor mouse
        }

    }

    //Proses Scan Original Image
    private ArrayList scan(Bitmap imgScr)
    {
        ArrayList pixel = new ArrayList();
        for (int j = 0; j < imgScr.Height; j = j + 3) // koordiant Y
        {
            for (int i = 0; i < imgScr.Width; i = i + 3) // koordinat X
            {
                if (imgScr.GetPixel(i, j) == cWhite)
                {
                    Vertex pix = new Vertex(i, j);
                    pixel.Add(pix);
                }
            }
        }
        return pixel;
    }


    //Mengolah Frame lebih lanjut
```

```
        private void FormEfek_KeyPress(object sender,
System.Windows.Forms.KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)Keys.F) //Tekan Shit+F untuk menampilkan full
screen
            {
                formstate.Maximize(this);
                //timer dijalankan saat full screen, saat ini gerakan cursor mulai diambil alih
oleh gerakan
                //yang terdeteksi oleh camera webcam
                timerCursor.Start();
            }
            else
                if (e.KeyChar == (char)Keys.Escape) // tekan Escape dan Form kembali
normal
                {
                    formstate.Restore(this);
                    //timer berhenti, mouse kembali berfungsi secara normal
                    timerCursor.Stop();
                }
        }
        //Timer Cursor dijalankan
        private void timerCursor_Tick(object sender, System.EventArgs e)
        {
            //posisi mouse ditentukan oleh posisi titik dari Vertex
            Cursor.Position = new Point(pointX * 2, pointY * 2);
        }

    }
}
```

B.5 Water Effect Picture Box

```
using System;
using System.Collections;
using System.Drawing;
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace Gerakan_Efek_Air
{
    public partial class WaterEffectPictureBox : Panel
```

```
{
   private Bitmap            bmp;
   private short[, ,]        waves;
   private int               waveWidth;
   private int               waveHeight;
   private int               activeBuffer = 0;
   private bool              weHaveWaves;
   private int               bmpHeight;
   private int               bmpWidth;
   private byte[]            bmpBytes;
   private BitmapData        bmpBitmapData;
   private int               scale;

   public WaterEffectPictureBox()
   {
      InitializeComponent();
      effecttimer.Enabled = true;
      effecttimer.Interval = 50;
      SetStyle(ControlStyles.UserPaint, true);
      SetStyle(ControlStyles.AllPaintingInWmPaint, true);
      SetStyle(ControlStyles.DoubleBuffer, true);
      this.BackColor = Color.White;
      weHaveWaves = false;
      scale = 1;
   }

   public WaterEffectPictureBox(Bitmap bmp) : this()
               {
                        this.ImageBitmap = bmp;
               }

   public Bitmap ImageBitmap
   {
      get { return bmp; }
      set
      {
         bmp = value;
         bmpHeight = bmp.Height;
         bmpWidth = bmp.Width;

         waveWidth = bmpWidth >> scale;
         waveHeight = bmpHeight >> scale;
```

```
            waves = new Int16[waveWidth, waveHeight, 2];

            bmpBytes = new Byte[bmpWidth * bmpHeight * 4];
            bmpBitmapData = bmp.LockBits(new Rectangle(0, 0, bmpWidth,
bmpHeight), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);
            Marshal.Copy(bmpBitmapData.Scan0, bmpBytes, 0, bmpWidth *
bmpHeight * 4);
        }
    }

    public int scale
    {
        get { return scale; }
        set { scale = value; }
    }

    private void effecttimer_Tick(object sender, System.EventArgs e)
    {

        if (weHaveWaves)
        {
            Invalidate();

            ProcessWaves();

        }
    }

    /// <summary>
    /// Paint handler
    ///
    /// Calculates the final effect-image out of
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    public void WaterEffectPictureBox_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
    {

        using (Bitmap tmp = (Bitmap)bmp.Clone())
        {
```

```
            int xOffset, yOffset;
            byte alpha;

            if (weHaveWaves)
            {
               BitmapData tmpData = tmp.LockBits(new Rectangle(0, 0, bmpWidth,
bmpHeight), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);

               byte[] tmpBytes = new Byte[bmpWidth * bmpHeight * 4];

               Marshal.Copy(tmpData.Scan0, tmpBytes, 0, bmpWidth * bmpHeight *
4);

               for (int x = 1; x < bmpWidth - 1; x++)
               {
                  for (int y = 1; y < bmpHeight - 1; y++)
                  {
                     int waveX = (int)x >> scale;
                     int waveY = (int)y >> scale;

                     //check bounds
                     if (waveX <= 0) waveX = 1;
                     if (waveY <= 0) waveY = 1;
                     if (waveX >= waveWidth - 1) waveX = waveWidth - 2;
                     if (waveY >= waveHeight - 1) waveY = waveHeight - 2;

                     //this gives us the effect of water breaking the light
                     xOffset = (waves[waveX - 1, waveY, activeBuffer] - waves[waveX
+ 1, waveY, activeBuffer]) >> 3;
                     yOffset = (_waves[waveX, waveY - 1, activeBuffer] -
waves[waveX, waveY + 1, activeBuffer]) >> 3;

                     if ((xOffset != 0) || (yOffset != 0))
                     {
                        //check bounds
                        if (x + xOffset >= bmpWidth - 1) xOffset = bmpWidth - x - 1;
                        if (y + yOffset >= bmpHeight - 1) yOffset = bmpHeight - y - 1;
                        if (x + xOffset < 0) xOffset = -x;
                        if (y + yOffset < 0) yOffset = -y;

                        //generate alpha
                        alpha = (byte)(200 - xOffset);
```

```
                        if (alpha < 0) alpha = 0;
                        if (alpha > 255) alpha = 254;

                        //set colors
                        tmpBytes[4 * (x + y * bmpWidth)] = bmpBytes[4 * (x + xOffset
+ (y + yOffset) * bmpWidth)];
                        tmpBytes[4 * (x + y * bmpWidth) + 1] = bmpBytes[4 * (x +
xOffset + (y + yOffset) * bmpWidth) + 1];
                        tmpBytes[4 * (x + y * bmpWidth) + 2] = bmpBytes[4 * (x +
xOffset + (y + yOffset) * bmpWidth) + 2];
                        tmpBytes[4 * (x + y * bmpWidth) + 3] = alpha;

                    }

                }
            }

            //copy data back
            Marshal.Copy(tmpBytes, 0, tmpData.Scan0, bmpWidth * bmpHeight *
4);
            tmp.UnlockBits(tmpData);

        }

        e.Graphics.DrawImage(tmp, 0, 0, this.ClientRectangle.Width,
this.ClientRectangle.Height);

    }

}

/// <summary>
/// This is the method that actually does move the waves around and simulates
the
/// behaviour of water.
/// </summary>
private void ProcessWaves()
{
    int newBuffer = (activeBuffer == 0) ? 1 : 0;
    bool wavesFound = false;

    for (int x = 1; x < waveWidth - 1; x++)
```

```
        {
            for (int y = 1; y < waveHeight - 1; y++)
            {
                _waves[x, y, newBuffer] = (short)(
                                ((waves[x - 1, y - 1, activeBuffer] +
                                waves[x, y - 1, activeBuffer] +
                                waves[x + 1, y - 1, activeBuffer] +
                                waves[x - 1, y, activeBuffer] +
                                waves[x + 1, y, activeBuffer] +
                                waves[x - 1, y + 1, activeBuffer] +
                                waves[x, y + 1, activeBuffer] +
                                waves[x + 1, y + 1, activeBuffer]) >> 2) - waves[x, y,
newBuffer]);

                //damping
                if (waves[x, y, newBuffer] != 0)
                {
                    waves[x, y, newBuffer] -= (short)(_waves[x, y, newBuffer] >> 4);
                    wavesFound = true;
                }


            }
        }

        weHaveWaves = wavesFound;
        activeBuffer = newBuffer;

    }

    /// <summary>
    /// This function is used to start a wave by simulating a round drop
    /// </summary>
    /// <param name="x">x position of the drop</param>
    /// <param name="y">y position of the drop</param>
    /// <param name="height">Height position of the drop</param>
    private void PutDrop(int x, int y, short height)
    {
        weHaveWaves = true;
        int radius = 20;
        double dist;
```

```
            for (int i = -radius; i <= radius; i++)
            {
                for (int j = -radius; j <= radius; j++)
                {
                    if (((x + i >= 0) && (x + i < waveWidth - 1)) && ((y + j >= 0) && (y + j
< waveHeight - 1)))
                    {
                        dist = Math.Sqrt(i * i + j * j);
                        if (dist < radius)
                            waves[x + i, y + j, activeBuffer] = (short)(Math.Cos(dist * Math.PI /
radius) * height);
                    }
                }
            }
        }

        /// <summary>
        /// The MouseMove handler.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void WaterEffectPictureBox_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e)
        {
            int realX = (int)((e.X / (double)this.ClientRectangle.Width) *
_waveWidth);
            int realY = (int)((e.Y / (double)this.ClientRectangle.Height) *
_waveHeight);
            PutDrop(realX, realY, 80);
        }
    }
}
```

B.6 Vertex

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace Gerakan_Efek_Air
{
```

```
public class Vertex
{
   ArrayList vertices = new ArrayList();

   //Inisialisasi x,y
   private int x;
   private int y;

   //Vertex adalah program untuk melalukan Looping pada pengambilan
   //data hasil scan, sehingga data dapat diambil secara kontinu
   public Vertex(int i, int j)
   {
      this.X = i;
      this.Y = j;
   }

   public void insert(Vertex v)
   {
      vertices.Add(v);
   }

   public ArrayList GetPixels()
   {
      return vertices;
   }
   public int GetCount()
   {
      return vertices.Count;
   }

   public void AddPixels(Vertex v)
   {
      vertices.Add(v);
   }
   public int X
   {
      get { return x; }
      set { x = value; }
   }

   public int Y
   {
```

```
            get { return y; }
            set { y = value; }
        }
    }
}
```

B.7 Form State

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace Gerakan_Efek_Air
{
    /// <summary>
    /// Selected Win AI Function Calls
    ///
    /// </summary>
    public class WinApi
    {
        [DllImport("user32.dll", EntryPoint = "GetSystemMetrics")]
        public static extern int GetSystemMetrics(int which);
        [DllImport("user32.dll")]
        public static extern void
        SetWindowPos(IntPtr hwnd, IntPtr hwndInsertAfter,
        int X, int Y, int width, int height, uint flags);
        private const int SM_CXSCREEN = 0;
        private const int SM_CYSCREEN = 1;
        private static IntPtr HWND_TOP = IntPtr.Zero;
        private const int SWP_SHOWWINDOW = 64; // 0x0040
        public static int ScreenX
        {
            get { return GetSystemMetrics(SM_CXSCREEN); }
        }

        public static int ScreenY
        {
            get { return GetSystemMetrics(SM_CYSCREEN); }
        }
        public static void SetWinFullScreen(IntPtr hwnd)
```

```
        {
            SetWindowPos(hwnd, HWND_TOP, 0, 0, ScreenX, ScreenY,
SWP_SHOWWINDOW);
        }
    }
    /// <summary>
    /// Class used to preserve / restore state of the form
    /// </summary>
    public class FormState
    {
        private FormWindowState winState;
        private FormBorderStyle brdStyle;
        private bool topMost;
        private Rectangle bounds;
        private bool IsMaximized = false;
        public void Maximize(Form targetForm)
        {
            if (!IsMaximized)
            {
                IsMaximized = true;
                Save(targetForm);
                targetForm.WindowState = FormWindowState.Maximized;
                targetForm.FormBorderStyle = FormBorderStyle.None;
                targetForm.TopMost = true;
                WinApi.SetWinFullScreen(targetForm.Handle);
            }
        }
        public void Save(Form targetForm)
        {
            winState = targetForm.WindowState;
            brdStyle = targetForm.FormBorderStyle;
            topMost = targetForm.TopMost;
            bounds = targetForm.Bounds;
        }
        public void Restore(Form targetForm)
        {
            targetForm.WindowState = winState;
            targetForm.FormBorderStyle = brdStyle;
            targetForm.TopMost = topMost;
            targetForm.Bounds = bounds;
            IsMaximized = false;
        }
```

```
    }
}
```

B. 8 Programs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Gerakan_Efek_Air
{
    static class Program
    {
        /// <summary>
        /// Saat Debug maka Form Efek yg akan di load
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new FormEfek());
        }
    }
}
```